C.A.R. HOARE
- Profesor în Teoria programării -
Universitatea Oxford

PROCESE SECVENȚIALE COMUNICANTE

Editura Fundației Chemarea IAȘI
1995

C.A.R. HOARE

– Profesor în Teoria programării –
Universitatea Oxford

PROCESE SECVENŢIALE COMUNICANTE

Traducere din limba engleză: ş.l. NICOLAE MARIAN
ş.l. DAN OLARU

Editura Fundaţiei Chemarea IAŞI
1995

Cuprins

Cuvânt înainte

Din mai multe motive, aceasta este o carte așteptată de toți cei care știau că se pregătește. A spune acum că răbdarea lor a fost pe deplin răsplătită ar fi o litotă.

Un simplu motiv face ca aceasta să fie prima carte a lui Tony Hoare. Mulţi îl cunosc din prelegerile pe care le-a ţinut neobosit peste tot şi mai mulţi îl ştiau ca autor atent şi pedant al unui număr de articole (de o mare varietate!) care au devenit clasice aproape înainte de a se usca cerneala. Dar o carte este altceva : aici autorul se poate exprima pe sine fără limitările obişnuite de timp şi spaţiu. De asemenea i se dă posibilitatea unei relevări proprii mai intime, acoperirii unei topici de o dimensiune mărită, ocazie pe care Tony Hoare a fructificat-o mai bine decât ne aşteptam.

Un motiv mai solid transpare însă din conţinutul cărţii. Cînd concurenţa şi-a făcut loc în conceptele de calcul acum un sfert de secol, a provocat o confuzie fără sfârşit, parţial datorată circumstanţelor diferite de unde provenea, parţial datorată introducerii nedeterminismului în acea perioadă. Descâlcirea acestei confuzii a necesitat munca încordată a unui profesionist devotat, care cu puţin noroc, putea clarifica situaţia. Tony Hoare şi-a dedicat o mare parte din activitatea ştiinţifică acestei provocări şi avem toate motivele să-i mulţumim.

Cel mai puternic motiv, totuşi, a fost subtil resimţit de acei care au recenzat manunscrisul, ce aruncă o nouă, surprinzătoare şi în acelaşi timp clară lumină asupra a ceea ce ştiinţa calculatoarelor poate – sau va – fi. A spune sau simţi că principalul obiectiv al unui savant din lumea calculatoarelor este de a nu crea confuzie prin propriile descoperiri este un aspect şi cu totul altceva este de a descoperi şi arăta cum eleganţa tangibilă şi aproape explicită a câtorva legi matematice călăuzitoare poate realiza o teorie sublimă. De aceea noi, cititorii recunoscători, vom alege după gust roadele inteligenţei ştiinţifice, tenacităţii şi agilităţii conceptuale ale lui C.A.R. Hoare.

Edsger W. Dijkstra

Prefață

Aceasta este o carte pentru informaticienii dornici de o mai bună înțelegere și perfecționare într-o profesie intelectuală dintre cele mai dinamice. Se adresează întâi unei curiozități firești provenită din noua tratare a unui domeniu cunoscut, fiind ilustrată de exemple selectate dintr-o gamă mare de aplicații : automate de vânzare, jocuri, sisteme de operare. Totul se bazează pe o teorie matematică descrisă sistematic prin legi algebrice. Obiectivul principal este de a introduce un punct de vedere ce permite cititorului să vadă problemele actuale și viitoare într-o lumină nouă în care rezolvările sunt mai sigure și eficiente sau, chiar mai mult, pot fi evitate neajunsurile.

Cea mai importantă aplicație a noilor concepte este specificarea, modelarea și implementare sistemelor care interacționează continuu cu mediul. Ideea de bază este posibilitatea descompunerii fără dificultate a acestor sisteme în subsisteme care operează concurent și interacționează între ele și mediu. Compunerea paralelă a subsistemelor este la fel de simplă ca și compunerea secvențială a instrucțiunilor în limbajele de programare convenționale.

Această privire interioară aduce și beneficii practice. Întâi evită multe din problemele paralelismului în programare – interferență, excludere mutuală, întreruperi, fire de execuție, semafoare etc. Apoi include drept cazuri particulare multe din ideile de structurare inovatoare care au fost dezvoltate prin cercetări recente în domeniul limbajelor de programare și metodologiei de programare – monitor, clasă, modul, pachet, regiune critică, plic, formă și chiar umila subrutină. În fine, asigură un fundament matematic sigur pentru evitarea erorilor ca divergență, blocaj, blocaj prin ciclare infinită și pentru realizarea unei corectitudini demonstrabile în modelarea și implementarea sistemelor informatice.

Am încercat din răsputeri să prezint ideile într-o ordine logică și psihologică strictă, pornind de la operatori de bază simpli și avansând către aplicații elaborate. Cititorii vor avea sigur un grad diferit de interes, pornind de la cei care vor citi cartea din scoarță în scoarță și până la cei interesați numai de anumite probleme. De aceea fiecare capitol al cărții este structurat pentru a permite o judicioasă selecție.

1. Fiecare idee nouă este mai întâi descrisă și ilustrată de un număr de exemple simple, care probabil vor fi de ajutor pentru toți cititorii.

2. Legile algebrice care descriu proprietățile esențiale ale diverșilor operatori vor interesa mai mult pe cei ce sunt atrași de eleganța matematică. Ele vor fi de ajutor și celor care doresc o optimizare a modelării prin utilizarea unor transformări echivalente cu păstrarea corectitudinii.
3. Implementările propuse sunt deosebite prin aceea că utilizează un subset minimal, funcțional, simplu al limbajului de programare LISP. Aceasta permite celor familiarizați cu LISP-ul să realizeze exerciții și demonstrații.
4. Definirile urmelor și specificațiilor sunt de interes pentru analiștii de sistem, care trebuie să exprime nevoile clientului înainte de a considera o implementare. Ele sunt de folos programatorilor experimentați, ce proiectează sisteme prin împărțirea în subsisteme cu specificarea clară a interfețelor.
5. Demonstrațiile interesează pe cei ce implementează, care trebuie să producă programe fiabile din specificațiile date, la termene fixate și cu prețuri de cost fixate.
6. În fine, teoria matematică dă o definiție riguroasă conceptului de proces și operatorilor, în contextul construirii proceselor. Aceste definiții sunt baza pentru legile algebrice, implementări și demonstrații.

Cititorul poate omite consecvent sau nu parcurgerea unora dintre aceste probleme, de un interes considerat mai scăzut sau cu un grad nai mare de dificultate.

Succesiunea capitolelor în carte a fost astfel organizată pentru a permite o judicioasă parcurgere, selecție sau rearanjare. Primele paragrafe din cap. 1 și 2 sunt o introducere necesară pentru toți cititorii. Celelalte paragrafe pot fi trecute în grabă sau amânate pentru o a doua lectură. Cap. 3, 4, 5 sunt independente unul de altul și pot fi începute în orice combinație corespunzător interesului și înclinației cititorului. Astfel, dacă se întâmpină o anumită dificultate de înțelegere într-o etapă, este preferabil să se continue cu paragraful următor sau cu capitolul următor, deoarece în mod normal materialul omis nu va fi necesar imediat. Când totuși o asemenea necesitate există, va fi în text o referință explicită înapoi, ce poate fi urmată sau nu. Sper ca totul în carte să fie interesant și să justifice efortul de a o citi. Ordinea parcurgerii ține de preferințele fiecăruia.

Exemplele alese pentru a ilustra ideile din carte vor părea foarte simple. Aceasta este și intenția. Primele exemple selectate pentru a ilustra o nouă idee trebuie să fie foarte simple pentru a nu umbri ideea prin complexitatea sau neobișnuitul lor. Simplitatea soluțiilor acestor exemple trebuie să se datoreze puterii conceptelor folosite și eleganței notațiilor

Cu toate acestea, fiecare cititor are problemele sale, uneori dureroase, mai complexe, mai importante decât exemplele alese. Aceste probleme par a nu putea fi rezolvate de vreo teorie matematică. Nu vă lăsați pradă disperării

și enervării și acceptați provocarea aplicării metodelor din această carte problemelor D-voastră. Începeți cu o variantă foarte simplificată a problemelor și adăugați mereu câte ceva până ajungeți la complexitatea cerută. Este surprinzător cum modelul simplificat poate fi o bază de pornire iar detaliile se pot suprapune fără afectarea siguranței. Surpriza finală este că unele detalii pot fi dovedite ca nefiind strict necesare, caz în care teoria își relevă întreaga sa putere.

După cum știm, notațiile sunt o permanentă problemă. Astfel, orice student pus să învețe limba rusă întâmpină dificultăți cu alfabetul chirilic, în special datorită pronunției. Ca o consolare, aceasta este o ultimă problemă ce s-ar întâlni aici. După învățarea simbolurilor se învață gramatica, vocabularul, idioamele, stilul și apoi se poate trece la fluientizarea exprimării ideilor. Toate acestea se fac însă treptat, cu exercițiu, și nu pot fi grăbite. Același lucru este și cu teoria matematică. Simbolurile apar la început ca o piedică, problema reală fiind de a învăța semnificația și proprietățile lor precum și posibilitățile de manipulare, pentru obținerea unei dexterități în exprimarea noilor soluții, demonstrații la diverse probleme. În fine, prin eleganța matematică și stilul folosit, simbolurile vor fi transparente, limpezindu-se și înțelesul lor. Avantajul covârșitor al matematicii este simplitatea regulilor față de limbajul natural și vocabularul redus. De asemenea, o soluție poate fi găsită prin deducție logică, prin consultarea unor cărți sau experți.

Din această cauză atât matematica cât și programarea sunt o permanentă provocare, nu totdeauna ușoară. Chiar matematicienii întâmpină dificultăți cu domenii noi. Teoria proceselor comunicante este un astfel de nou domeniu, iar programatorii care îl abordează nu trebuie să aibă nici un handicap față de matematicieni, din contră, au avantajul de a-și pune cunoștințele în practică. Materialul acestei cărți provine din prezentări în simpozioane, prelegeri și cursuri academice. El a fost proiectat la început ca un curs universitar de un semestru în inginerie software, postgraduate, cu toate că cea mai mare parte poate fi prezentat în a doua jumătate a anului întâi, sau prima parte a anului doi, undergraduate, secția calculatoare. Cerințele minime sunt de algebră superioară de liceu, concepte din teoria mulțimilor și calculul predicatelor. Cartea este adecvată și pentru un curs intensiv de o săptămână pentru programatori experimentați. Într-un astfel de curs, lectorul ar trebui să se concentreze pe exemple și definiții, lăsând la o parte aparatul matematic pentru studiul individual. Este posibil și prezentare sub formă de curs numai a primelor două capitole, sau într-o singură prelegere a problemei celor cinci filozofi.

Nu este deloc ușor de ținut cursuri și seminarii în domeniul proceselor secvențiale comunicante. Exemplele ce trebuie folosite constituie mereu o problemă, dacă punem la socoteală numai gradul de percepție al participanților, sensibilitatea lor. Având în vedere eleganța abstractă a formulelor matematice, unele concepte prezentate în carte au ceva asemănător unor 'fugi' de J. S. Bach.

Sumar

Cap. 1 introduce conceptele de bază ale proceselor și abstractizează matematic interacțiunile dintre sistem și mediu. Se arată cum familiara tehnică a recursivității poate fi folosită pentru a descrie procese de o durată variabilă sau chiar infinită. Conceptele sunt explicate întâi prin exemple și grafic. Explicația completă o oferă legile algebrice iar implementarea se face în limbajul funcțional LISP. A doua parte a capitolului explică cum comportarea unui proces poate fi înregistrată ca urma secvențelor sale de acțiuni în care a fost implicat. Sunt definite multe operații utile cu urme. Un proces poate fi specificat înaintea implementării prin descrierea proprietăților urmelor sale. Sunt date reguli ce ajută la implementarea proceselor ce se pot dovedi corespunzătoare specificațiilor.

Al doilea capitol prezintă cum procesele pot fi asamblate în sisteme, în care componentele interacționează unele cu altele sau cu mediul extern. Introducerea concurenței nu introduce automat un factor de nedeterminism. Principalul exemplu din acest capitol este celebra problemă a celor cinci filozofi. A doua parte a cap. 2 prezintă adaptarea proceselor la situații noi prin schimbarea numelui evenimentelor în care sunt antrenate. Capitolul se încheie cu expunerea teoriei matematice a proceselor deterministe, incluzând elemente de teoria recursivității.

Al treilea capitol furnizează una din cele mai simple și cunoscute soluții pentru nedeterminism. Nedeterminismul este prezentat ca o tehnică de pătrundere a necunoscutului, din moment ce el rezultă natural din decizia de a ignora sau trece sub tăcere acele aspecte ale comportării sistemului de care nu suntem interesați. De asemenea se păstrează anumite simetrii cu determinismul în definiția operatorilor din teoria matematică. Demonstrațiile pentru procesele nedeterministe sunt ușor mai complicate decât cele de la procese deterministe, deoarece trebuie arătat că orice alegere posibil nedeterministă va avea o comportare corespunzătoare specificațiilor date. Din fericire, există tehnici pentru evitarea nedeterminismului și ele sunt folosite exclusiv în cap. 4 și 5. Astfel, studiul cap. 3 poate fi amânat până înainte de cap. 6, când introducerea nedeterminismului nu mai poate fi amânată.

În ultimele paragrafe ale cap. 3 se prezintă complet definirea matematică a conceptului de proces nedeterminist. Această parte va fi de interes pentru matematicianul care intenționează să exploreze fundamentelor subiectului sau

să verifice validitatea demonstraţiilor legilor algebrice şi a celorlalte proprietăţi ale proceselor. Programatorii pot privi legile ca ceva evident sau justificat de utilitatea lor, omiţând fără probleme porţiunile mai teoretice.

Cap. 4 introduce în sfârşit comunicarea : un caz special de interacţiune a două procese, când unul emite un mesaj pe care celălalt, în acelaşi timp, îl recepţionează. Avem astfel de a face cu o comunicare sincronizată. Dacă este necesară bufferarea pe un canal, aceasta se realizează prin interpunerea unui proces buffer între cele două procese. Un obiectiv major în proiectarea sistemelor concurente este viteza de răspuns în soluţionarea problemelor practice. Aceasta este ilustrat prin modelarea unor algoritmi sistolici sau iterativi. Un caz simplu de comunicaţie este o conductă, definită ca o secvenţă de procese în care fiecare primeşte informaţii de la predecesor şi transmite informaţii la succesor. Conductele sunt utile pentru implementarea unor protocoale de comunicaţie unidirecţionale, structurate ca o ierarhie de nivele. În fine, importantul concept de tip abstract de dată este modelat cu un proces subordonat, fiecare instanţă a lui comunicând numai cu blocul în care este declarat.

În cap. 5 se prezintă operatorii convenţionali din programarea secvenţială şi încadrarea lor în modelul proceselor secvenţiale comunicante. Poate apărea surprinzător pentru programatorii experimentaţi că aceşti operatori se bucură de acelaşi fel de elegante proprietăţi algebrice ca şi operatorii teoriilor matematice familiare, putându-se dovedi că programele secvenţiale respectă specificaţiile aproape la fel ca programele concurente, fiind definită chiar şi o întrerupere externă pentru a fi utilizabilă în legile folosite.

Cap. 6 descrie cum se structurează şi implementează un sistem în care un număr limitat de resurse fizice, ca de exemplu discurile sau imprimantele, pot fi partajate de mai multe procese, ale căror necesităţi de resurse variază in timp. Fiecare resursă este reprezentată ca un singur proces. De fiecare dată când o resursă este cerută de un proces utilizator, o nouă resursă virtuală este creată. O resursă virtuală se comportă ca şi când ar fi subordonată procesului utilizator, comunicând cu resursa reală ori de câte ori e necesar. Aceste comunicaţii sunt înteţesute cu acelea ale altor procese concurente active virtuale. Astfel procesele reale şi virtuale joacă acelaşi rol ca şi monitoarele şi plicurile din Pascal Plus. Capitolul este ilustrat de o dezvoltare modulară şi gradată a unor sisteme de operare atât simple cât şi complete, cele mai mari exemple de altfel din carte.

Cap. 7 descrie un număr de alternative la concurenţă şi comunicaţie care există deja şi explică din punct de vedere practic, istoric şi personal motivele care au dus la teoria expusă în capitolele precedente. Aici se recunoaşte influenţa altor autori (lucrări), se dau recomandări şi o introducere la lecturi următoare în domeniu.

Aprecieri

Este o mare plăcere să recunosc munca profundă şi originală a lui Robin Milner, expusă în lucrarea sa clasică *Calculul proceselor comunicante*. Ideile sale originale, prietenia sa şi rivalitatea profesională au fost surse constante de inspiraţie şi încurajare pentru eforturile ce au culminat cu publicarea acestei cărţi.

În ultimii douăzeci de ani m-au preocupat problemele legate de programarea concurentă şi proiectarea limbajelor de programare. În această perioadă am beneficiat de colaborarea cu mari savanţi ca Per Brinch Hansen, Stephen Brookes, Dave Bustard, Zhou Chao Chen, Ole-Johan Dahl, Edsger W. Dijkstra, John Elder, Jeremy Jacob, Ian Hayes, Jim Kaubisch, John Kennaway, T. Y. Kong, Peter Lauer, Mike McKeag, Carroll Morgan, Ernst-Rudiger Olderog, Rudi Reinecke, Bill Roscoe, Alex Teruel, Alastair Tocher şi Jim Welsh.

În final, mulţumesc în mod deosebit lui O.-J. Dahl, E. W. Dijkstra, Leslie M. Goldschlager, Jeff Sanders şi altora care cu migală au studiat acest text şi-au scos la iveală erori sau obscurităţi. De asemenea mulţumesc participanţilor la şcoala de vară din Wollongong, secţia programare, ianuarie 1983, studenţilor din Graduate School of Chinese Academy of Science, aprilie 1983 şi studenţilor postgraduate în informatică din Oxford University, anii 1979-1984.

Glosar de simboluri

Logică

Notații	*Semnificație*	*Exemple*
$=$	egal	$x=x$
$\neq$	diferit de	$x\neq x+1$
$\square$	sfârşitul unui exemplu sau demonstraţie	
$P\wedge Q$	P şi Q (conjuncţie)	$x\leq x+1\wedge x\neq x+1$
$P\vee Q$	P sau Q (disjuncţie)	$x\leq y\vee y\leq x$
$\neg P$	P negat (negaţie)	$\neg 3\geq 5$
$P\Rightarrow Q$	dacă P atunci Q	$x<y\Rightarrow x\leq y$
$P\equiv Q$	P dacă şi numai dacă Q	$x<y\equiv y>x$
$\exists x.P$	există un x cu proprietatea P	$\exists x.x>y$
$\forall x.P$	pentru toţi x cu proprietatea P	$\forall x.x<x+1$
$\exists x{:}A.P$	există un x din mulţimea A cu proprietatea P	
$\forall x{:}A.P$	pentru toţi x din A cu proprietatea P	

Mulţimi

Notații	*Semnificație*	*Exemple*
$\in$	aparţine	Napoleon $\in$umanităţii
$\notin$	nu aparţine	Napoleon $\notin$ ruşilor
$\{\}$	mulţime vidă	$\neg$ Napoleon $\in$ $\{\}$
$\{a\}$	mulţime cu un singur element (singleton)	$x\in\{a\}\equiv x=a$
$\{a,b,c\}$	mulţime cu trei elemente	$c\in\{a,b,c\}$

$\{x \mid P(x)\}$	mulţimea tuturor x cu proprietatea P	$\{a\}=\{x \mid x=a\}$
$A \cup B$	A reunit cu B	$A \cup B=\{x \mid x \in A \vee x \in B\}$
$A \cap B$	A intersectat cu B	$A \cap B=\{x \mid x \in A \wedge x \in B\}$
$A-B$	A minus B	$A-B=\{x \mid x \in A \wedge \neg x \in B\}$
$A \subseteq B$	A este inclus în B	$A \subseteq B \equiv \forall x{:}A.x \in B$
$A \supseteq B$	A include B	$A \supseteq B \equiv B \subseteq A$
$\{x{:}A \mid P(x)\}$	mulţimea x din A cu $P(x)$	
$\mathbf{N}$	mulţimea numerelor naturale	$\{0,1,2,...\}$
$\mathbf{P}A$	mulţimea părţilor lui A	$\mathbf{P}A=\{X \mid X \subseteq A\}$
$\bigcup_{n\geq 0} A_n$	reuniunea unei familii de mulţimi	$\bigcup_{n\geq 0} A_n=\{x \mid \exists n\geq 0.x \in A\}$
$\bigcap_{n\geq 0} A_n$	intersecţia unei familii de mulţimi	$\bigcap_{n\geq 0} A_n=\{x \mid \forall n\geq 0.x \in A\}$

Funcţii

Notaţii	*Semnificaţie*	*Exemple*
$f{:}A \to B$	f este definită pe A cu valori în B	pătrat: $\mathbf{N} \to \mathbf{N}$
$f(x)$	un element din B ce corespunde lui x din A	
injecţie	o funcţie ce face să-i corespundă fiecărui element din A un element distinct din B	$x \neq y \Rightarrow f(x) \neq f(y)$
f^{-1}	inversa unei injecţii f	$x=f(y) \equiv y=f^{-1}(x)$
$\{f(x) \mid P(x)\}$	mulţimea tuturor imaginilor lui x cu $P(x)$	$\{y \mid \exists x.y=f(x),P(x)\}$
$f(C)$	imaginea lui C prin f	pătrat($\{3,5\}$)=$\{9,25\}$
$f \circ g$	f compus cu g	$f \circ g(x)=f(g(x))$
$\lambda x.f(x)$	funcţia care face să-i corespundă fiecărei valori a lui x un $f(x)$	$(\lambda x.f(x))(3)=f(3)$

Urme

Paragraf	*Notaţii*	*Semnificaţie*	*Exemple*
1.5	$\langle\rangle$	urmă vidă	
1.5	$\langle a\rangle$	urmă cu element a	singleton
1.5	$\langle a,b,c\rangle$	urmă cu trei elemente	a apoi b apoi c
1.6.1	$\wedge$	concatenarea	$\langle a,b,c\rangle=\langle a,b\rangle\wedge\langle\rangle\wedge\langle c\rangle$

1.6.1	s^n	*s* repetată de *n* ori	$<a,b>^2=<a,b,a,b>$
1.6.2	$s\restriction A$	*s* restricţionată la *A*	$<b,c,d,a>\restriction\{a,c\}=<c,a>$
1.6.5	$s\leq t$	*s* este un prefix al lui *t*	$<a,b>\leq<a,b,c>$
4.2.2	$s\overset{n}{\leq} t$	*s* coincide cu *t* mai puţin *n* simboluri	$<a,b>\overset{2}{\leq}<a,b,d,c>$
1.6.5	*s* in *t*	*s* este conţinut în *t*	$<c,d>$ in $<b,c,d,a,b>$
1.6.6	$\#s$	lungimea lui *s*	$\#<b,c,b,a>=4$
1.6.6	$s\downarrow b$	numărul de simboli *b* în *s*	$<b,c,b,a>\downarrow b=2$
1.9.6	$s\downarrow c$	comunicaţia pe canalul *c* înregistrată în *s*	$<c.1,a.4,c.3,d.1>\downarrow c$ $=<1,3>$
1.9.2	$\hat{}/s$	*s* reunit	$\hat{}/<<a,b>,<>,<c>>$ $=<a,b,c>$
1.9.7	$s;t$	*s* terminată cu succes şi urmată de *t*	$(s\hat{}<\surd>);t=s\hat{}t$
1.6.4	A^*	mulţimi de secvenţe cu elemente din *A*	$A^*=\{s \mid s\restriction A=s\}$
1.6.3	s_0	capul secvenţei *s*	$<a,b,c>_0=<a>$
1.6.3	s'	coada secvenţei *s*	$<a,b,c>'=<b,c>$
1.9.4	$s[i]$	elementul *i* din *s*	$<a,b,c>[1]=b$
1.9.1	$f^*(s)$	*f* stea de *s*	pătrat$^*(<1,5,3>)$ $=<1,25,9>$
1.9.4	$\bar{s}$	inversa lui *s*	$\overline{\langle a,b,c\rangle}=<c,b,a>$

Evenimente speciale

Paragraf	*Notaţii*	*Semnificaţie*
1.9.7	$\surd$	succes (terminare cu succes)
2.6.2	*l.a*	participarea la evenimentul *a* a procesului etichetat *l*
4.1	$c.v$	comunicarea valorii v pe canalul *c*
4.5	*l.c*	canalul *c* a procesului etichetat *l*
4.5	$l.c.v$	comunicarea mesajului v pe canalul *l.c*
5.4.1	↯	catastrofă
5.4.3	$\otimes$	interschimbarea
5.4.4	©	punct de control pentru reluări
6.2	*ocupă*	acapararea unei resurse
6.2	*eliberează*	eliberarea unei resurse



Procese

Paragraf	*Notații*	*Semnificație*
1.1	αP	alfabetul procesului *P*
4.1	αc	mulțimea mesajelor comunicabile pe canalul *c*
1.1.1	$a \to P$	*a* apoi *P*
1.1.3	$(a \to P \mid b \to Q)$	*a* apoi *P* sau (alternativă) *b* apoi *Q* (dacă $a \neq b$)
1.1.3	$(x{:}A \to P(x))$	(alegerea lui *x*) din *A* apoi *P*(*x*)
1.1.2	$\mu X{:}A.F(X)$	procesul *X* cu alfabetul *A* astfel că $X=F(X)$
1.8.3	P/s	*P* după (angajarea în evenimentele urmei) *s*
2.3	$P \| Q$	*P* în paralel cu *Q*
2.6.2	$l{:}P$	*P* cu eticheta *l*
2.6.4	$L{:}P$	*P* cu etichete din mulțimea *L*
3.2	$P \sqcap Q$	*P* sau *Q* (nedeterminist)
3.3	$P \square Q$	*P* sau *Q*
3.5	$P \setminus C$	*P* fără *C* (mascare)
3.6	$P \| \| Q$	*P* întrețesut cu *Q*
4.4	$P \gg Q$	*P* înlănțuit cu *Q*
4.5	$P /\!/ Q$	*P* subordonat lui *Q*
6.4	$l{::}p /\!/ Q$	subordonare la distanță
5.1	$P;Q$	*P* (terminat cu succes) urmat de *Q*
5.4	$P \hat{} Q$	*P* întrerupt de *Q*
5.4.1	$P ↯ Q$	*P* sau în caz de catastrofă *Q*
5:4.2	$\hat{P}$	*P* restartabil
5.4.3	$P \otimes Q$	*P* în alternanță cu *Q*
5.5	$P \not< b \not> Q$	*P* dacă *b* altfel *Q*
5.1	$*P$	*P* repetat
5.5	$b * P$	cât timp *b* repetă *P*
5.5	$x := e$	*x* devine (valoarea) *e*
4.2	$b!e$	pe (canalul) *b* se emite (valoarea) *e*
4.2	$b?x$	din (canalul) *b* se preia *x*
6.2	$l!e?x$	apelul unei subrutine partajate cu numele *l* valoarea parametrului *e* și rezultatul în *x*
1.10.1	*P* **sat** *S*	(procesul) *P* satisface (specificația) *S*
1.10.1	*ur*	o urmă arbitrară a procesului specificat
3.7	*ref*	un refuz arbitrar al procesului specificat
5.5.2	$\grave{x}$	valoarea finală a lui *x* produsă de procesul specificat
5.5.1	*var*(*P*)	mulțimea variabilelor asignabile în *P*
5.5.1	*acc*(*P*)	mulțimea variabilelor accesibile în *P*

2.8.2	$P \sqsubseteq Q$	(determinist) Q poate face cel puţin cât şi P
3.9	$P \sqsubseteq Q$	(nedeterminist) Q este cel puţin la fel de bun ca P sau mai bun
5.5.1	$\mathcal{D}e$	expresia e este definită

Algebră

Termen	*Semnificație*	
reflexiv	o relaţie R astfel că	xRx
antisimetric	o relaţie R astfel că	$xRy \wedge yRx \Rightarrow x=y$
tranzitivă	o relaţie R astfel că	$xRy \wedge yRz \Rightarrow xRz$
ordine parţială	o relaţie $\leq$ care este reflexivă, antisimetrică şi tranzitivă	
limită	un cel mai mic element astfel că	$\bot \leq x$
monotonă	o funcţie f care respectă o ordine parţială	$x \leq y \Rightarrow f(x) \Rightarrow f(y)$
strictă	o funcţie identitate asupra limitei	$f(\bot)=\bot$
idempotent	un operator binar f astfel că	$xfx=x$
simetric	un operator binar f astfel că	$xfy=yfx$
asociativ	un operator binar f astfel că	$xf(yfz)=(xfy)fz$
distributivitate	f se distribuie cu g dacă	$xf(ygz)=(xfy)g(xfz) \wedge (ygz)fx=(yfx)g(zfx)$
unitate	lui f este un element 1 astfel că	$xf1=1fx=x$
zero	lui f este un element 0 astfel că	$xf0=0fx=0$

Grafuri

Termen	*Semnificație*
graf	o relaţie reprezentată printr-un desen
nod	un cerculeţ în graf reprezentând un element din domeniul de definiţie sau de valori
arc	o linie simplă sau cu o săgeată într-un graf ce conectează două noduri între care are loc relaţia
graf neorientat	graf cu o relaţie simetrică
graf orientat	graf al unei relaţii nesimetrice reprezentat prin arce cu săgeţi
ciclu orientat	o mulţime de noduri legate prin arce cu săgeţi orientate în acelaşi sens

ciclu neorientat	**o mulţime de noduri legate prin arce simple sau cu săgeţi în sensuri contrare**

Procese

1.1 Introducere

Să uităm puţin modul cum privim calculatoarele, programarea şi să ne gândim la obiectele din lumea înconjurătoare, care interacţionează între ele şi cu noi sau au diferite reguli de comportare. Să ne gândim la ceasuri, numărătoare, telefoane, jocuri electronice, automate de vânzare. Pentru a descrie regulile lor de comportare trebuie mai întâi să decidem ce fel de evenimente sau acţiuni ne interesează. Totodată să alegem un nume diferit pentru fiecare tip. În cazul unui automat de vânzare simplu există două feluri de evenimente :

mon introducerea unei monezi în automat
choc extragerea unei ciocolate din aparat

În cazul unui automat mai complex pot fi deosebite o varietate de evenimente:

*in*1*p* introducerea unei monezi de un penny
*in*2*p* introducerea unei monezi de doi penny
mic extragerea unui biscuit sau napolitane mici
mare extragerea unui biscuit sau napolitane mari
*ex*1*p* extragere restului de un penny

Înţelegem că fiecare nume de eveniment cuprinde o *clasă* de evenimente, adică apariţiile lui în timp. Distincţia între clasă şi apariţie este analoagă, de exemplu, în cazul literei c, ce apare, evident, de foarte multe ori în acestă carte.

Mulţimea numelor evenimentelor ce sunt considerate relevante pentru o anumită descriere a unui obiect se numeşte *alfabet*. Alfabetul este o proprietate apriori şi permanentă a unui obiect. Din punct de vedere logic este imposibil pentru un obiect să se angajeze într-un eveniment ce nu face parte din alfabetul său. De exemplu, un automat de vândut ciocolată nu va putea dintr-o dată să vândă jucării. S-ar putea ca automatul să nu furnizeze ciocolată

deoarece este defect, sau ciocolata s-a terminat, sau nimeni nu doreşte să o cumpere. Atât timp însă cât alfabetul său conţine evenimentul *choc*, alfabetul rămâne neschimbat chiar dacă evenimentul nici măcar nu apare.

Alegerea unui alfabet presupune o etapă deliberată de simplificare, de ignorare a proprietăţilor şi acţiunilor considerate de un interes mai scăzut. De exemplu, culoarea, greutatea, forma automatului nu se consideră relevante, de asemenea alimentarea cu ciocolată sau golirea casetei de monezi, sunt considerate fără interes pentru cumpărător.

Apariţia fiecărui eveniment în existenţa unui obiect se consideră o acţiune instantanee, atomică, fără durată. Acţiuni extinse sau consumatoare de timp pot fi reprezentate ca o pereche de evenimente, primul corespunzător începutului duratei de timp iar al doilea corespunzător sfârşitului ei. Durata unei acţiuni este dată de intervalul dintre apariţiile evenimentelor de început şi de sfârşit. În timpul unui astfel de interval pot apărea diverse alte evenimente. Două acţiuni extinse se pot suprapune în timp dacă începutul uneia precede sfârşitul celeilalte.

Alt detaliu ignorat deliberat este momentul apariţiei evenimentelor. Avantajul rezidă în simplificarea proiectării şi a raţionamentelor, ele putându-se aplica pentru sisteme software în orice context de viteză sau performanţe. În cazul în care momentele apariţiilor sunt critice, ele pot fi tratate independent de corectitudinea logică a modelului. De altfel, independenţa de timp a fost o condiţie permanent necesară pentru succesul limbajelor de programare de nivel înalt.

O consecinţă a ignorării timpului este refuzul considerării simultaneităţii evenimentelor. Când devine totuşi esenţială simultaneitatea unei perechi de evenimente, (ca de ex. în sincronizare) perechea se reprezintă printr-un singur eveniment, în celelalte cazuri perechea reprezentându-se prin două evenimente, ce se succed într-o ordine dată.

La constituirea alfabetului nu se va face distincţie între evenimentele ce ţin de obiect (de ex. *choc*) şi cele care ţin de un agent exterior obiectului (de ex. *mon*). Evitarea cauzalităţii conduce astfel la o semnificativă simplificare a teoriei şi a exemplificărilor ei.

Din acest moment vom folosi noţiunea de *proces* pentru a defini comportarea unui obiect având în vedere evenimentele ce compun alfabetul său. Vom folosi următoarele convenţii :

1. Evenimentele vor fi reprezentate prin cuvinte scrise cu litere mici, de exemplu : *mon*, *choc*, *in*2*p*, *ex*1*p*, sau pur şi simplu prin litere mici, de exemplu : *a*, *b*, *c*, *d*, *e*.
2. Procesele vor fi reprezentate prin cuvinte scrise cu majuscule, de exemplu : *AVS* automat de vândut simplu *AVC* automat de vândut complex sau prin majuscule simple, de exemplu : *P*, *Q*, *R*, ce desemnează în legi procese arbitrare.

3. Literele *x*, *y*, *z* sunt variabile cu semnificație de evenimente.
4. Literele *A*, *B*, *C* corespund unor mulțimi de evenimente (alfabete).
5. Literele *X*, *Y* sunt variabile ce semnifică procese.
6. Alfabetul unui proces este notat αP, de exemplu :

αAVS={*mon*, *choc*}
αAVC={*in*1*p*, *in*2*p*, *mic*, *mare*, *ex*1*p*}

Procesul cu alfabetul *A* care nu se angajează în nici un eveniment din alfabetul său se numește $STOP_A$. Acest proces descrie comportarea unui obiect care și-a terminat activitatea. Cu toate că există evenimente în alfabetul său, deci ar exista posibilitatea angajării în unele dintre ele, acest lucru nu se mai întâmplă. De asemenea, obiecte cu alfabete diferite devin diferite, independent de angajarea în evenimente. Astfel $STOP_{\alpha AVS}$ este diferit de $STOP_{\alpha AVC}$, deoarece alfabetele lor nu sunt identice.

Vom trece la definirea unor notații simple pentru a putea descrie obiecte ce se angajează în diverse evenimente.

1.1.1 Prefix

Fie *x* un eveniment și *P* un proces. Atunci

$(x \rightarrow P)$ (semnifică "*x* apoi *P*")

descrie un obiect care întâi se angajează în evenimentul *x* și apoi se comportă ca și *P*. Procesul $STOP_{\alpha AVS}$ este definit cu același alfabet ca și *P*, de aceea *x* trebuie să aparțină neapărat alfabetului (αP), sau formal

$\alpha(x \rightarrow P) = \alpha P$ dacă $x \in \alpha P$

Exemple

X1 Un automat de vânzare simplu care primește o monedă și apoi se strică ("o înghite")

$(mon \rightarrow STOP_{\alpha AVS})$ □

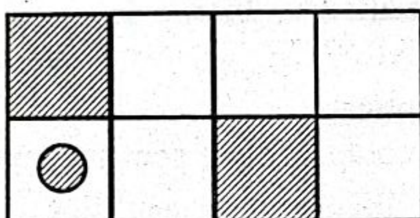
X2 Un automat simplu care este folosit cu succes de doi clienți înainte de a se strica

$(mon \rightarrow (choc \rightarrow (mon \rightarrow (choc \rightarrow STOP_{\alpha AVS}))))$

După cum se observă, *AVS* va accepta o monedă după care va furniza o ciocolată, iar va accepta o monedă şi iar va furniza o ciocolată, în final oprindu-se. Automatul nu va putea accepta două monezi consecutiv, furnizând două ciocolate (rezultă din funcţionarea automatului care blochează mecanic introducerea celei de-a doua monezi până la eliberarea ciocolatei pentru moneda anterioară). □

Pe viitor se va renunţa la paranteze în cazul unei succesiuni de evenimente ca acelea din X2, cu convenţia că $\rightarrow$ este asociativ la dreapta.

X3 Un marker, pornind din colţul stânga jos al unei suprafeţe riglate, se poate mişca numai la dreapta sau în sus într-o căsuţă adiacentă



$\alpha MSR=\{sus, dreapta\}$

$MSR=(dreapta \rightarrow sus \rightarrow dreapta \rightarrow dreapta \rightarrow STOP_{\alpha MSR})$ □

Trebuie observat că operatorul $\rightarrow$ presupune un proces în dreapta şi *un singur eveniment* în stânga. Astfel, dacă P şi Q sunt procese este incorect sintactic

$P \rightarrow Q$

Metoda prin care se descrie un proces care iniţial se comportă ca P şi apoi ca Q se va prezenta în cap. 4. De asemenea, dacă x şi y sunt evenimente, este incorect sintactic

$x \rightarrow y$

Putem avea însă pentru un proces

$x \rightarrow (y \rightarrow STOP)$

Trebuie să distingem cu atenţie conceptul de eveniment de acela de proces care se angajează în evenimente – pot fi mai multe sau nici unul.

1.1.2 Recursivitatea

Notația prefix poate fi folosită pentru a descrie comportarea completă a unui proces care eventual se și oprește. Dar devine anevoios de a descrie comportarea unui automat care funcționează ciclic. De aceea avem nevoie să descriem tipuri de comportări repetitive printr-o notație mult mai scurtă. Aceste notații nu ar trebui să țină cont de durata comportării ciclice a obiectului. Considerăm un ceas care ticăie

$\alpha CEAS=\{tic\}$

Să considerăm un obiect ceas care se comportă exact ca ceasul inițial, dar emite mai întâi un *tic*

$(tic \rightarrow CEAS)$

Evident comportarea acestui obiect nu poate fi deosebită de cea a ceasului original. Acest raționament ne conduce la ecuația

$CEAS=(tic \rightarrow CEAS)$

Această ecuație poate fi privită ca definind comportarea unui ceas, în același mod în care rădăcina pătrată din doi poate fi privită ca fiind soluția pozitivă a ecuației

$x=x^2+x-2$

Se pot scrie ecuații obținute din cea inițială prin substituirea în dreapta a membrului stâng

$CEAS=(tic \rightarrow CEAS)$	ecuație inițială
$=(tic \rightarrow (tic \rightarrow CEAS))$	substituție
$CEAS=(tic \rightarrow tic \rightarrow tic \rightarrow CEAS)$	similar

După cum se observă, membrul drept poate fi desfășurat cât vrem, având de-a face cu o comportare nemărginită

$tic \rightarrow tic \rightarrow tic \rightarrow \ldots$

la fel cum rădăcina pătrată din doi poate fi văzută ca limită a numărului de zecimale

1,414...

Descrierea recursivă de mai sus va fi efectivă numai dacă membrul drept începe cu cel puţin un eveniment prefixat tuturor apariţiilor recursive ale numelui procesului. De exemplu, ecuaţia recursivă

$$X=X$$

nu este efectivă din punctul nostru de vedere, deoarece orice proces poate fi o soluţie a sa. O descriere a unui proces care începe cu un prefix se spune că este cu *gardă*. Dacă $F(X)$ este o expresie cu gardă conţinând numele procesului X, iar A este alfabetul său, ecuaţia

$$X=F(X)$$

are soluţie unică dându-se alfabetul A. Vom nota soluţia prin expresia

$$\mu X{:}A.F(X)$$

X poate fi privit ca o variabilă locală şi prin urmare poate fi schimbat

$$\mu X{:}A.F(X)=\mu Y{:}A.F(Y)$$

Această egalitate este justificată de faptul că o soluţie pentru X din ecuaţia

$$X=F(X)$$

este şi o soluţie a ecuaţiei

$$Y=F(Y)$$

Pe viitor, vom da definirile recursive ale proceselor atât cu ajutorul ecuaţiilor, cât şi cu ajutorul lui μ, evident mai convenabil. În cazul că folosim varianta cu $\mu X{:}A.F(X)$, vom omite menţionarea explicită a alfabetului A, dacă acesta este evident din contextul sau conţinutul procesului.

Exemple

X1 Funcţionarea unui ceas va fi descrisă de

$$CEAS=\mu X{:}\{tic\}.(tic\rightarrow X)$$ □

X2 Un automat de vândut ciocolată care funcţionează non-stop

$AVS=(mon \rightarrow (choc \rightarrow AVS))$

După cum s-a menţionat mai sus, această ecuaţie este o altă formă de scriere pentru definirea cea mai formală

$AVS=\mu X:\{mon,choc\}.(mon \rightarrow (choc \rightarrow X))$ □

X3 Un automat care schimbă o monedă de cinci penny în monezi de un penny şi doi penny

$\alpha SCH5A=\{in5p,ex1p,ex2p\}$
$SCH5A=(in5p \rightarrow ex2p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5A)$ □

X4 Tot un automat de schimbat monezi, dar cu altă succesiune

$SCH5B=(in5p \rightarrow ex1p \rightarrow ex1p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5B)$ □

Faptul că ecuaţiile cu gardă au o soluţie şi ea este unică poate fi arătat prin metoda substituţiei. De fiecare dată când membrul drept al ecuaţiei este substituit numelui procesului, se extinde definirea comportării oricât de mult, astfel că orice comportare finită poate fi determinată în acest mod. Putem extrapola prin urmare, că două procese care au aceeaşi comportare până la un moment oarecare finit de timp, sunt de fapt acelaşi proces. O demonstraţie formală, completă, se va face odată cu definirea matematică a proceselor, dată în 2.8.3. Se va da şi o semnificaţie pentru recursivitatea fără gardă în 3.8.

1.1.3 Alegere

După cum s-a văzut, cu ajutorul recursivităţii şi a prefixului este posibil de a descrie obiecte cu un singur fir de comportare, fără ramificaţii. Totuşi, multe obiecte interacţionează cu mediul, ceea ce se reflectă asupra comportării lor prin aceea că trebuie făcută o alegere la un moment dat. De exemplu, un automat de vânzare poate oferi două canale de introdus monezi, unul pentru un penny, celălalt pentru doi penny. Cumpărătorul trebuie să decidă angajarea în unul din cele două evenimente. Dacă x şi y sunt două evenimente distincte

$(x \rightarrow P | y \rightarrow Q)$

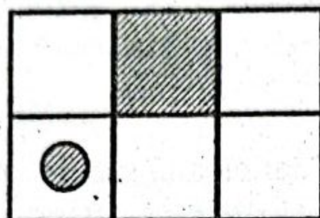
descrie un obiect care se angajează fie în evenimentul x fie în y. Comportarea ulterioară a obiectului este descrisă de P, dacă evenimentul care a avut loc a fost x, sau de Q, dacă evenimentul care a avut loc a fost y. Deoarece x şi y sunt evenimente diferite, alegerea între P şi Q este determinată de primul care a avut loc. Referitor la alfabete avem

$\alpha(x \to P | y \to Q) = \alpha P$ dacă $\{x, y\} \subseteq \alpha P$ şi $\alpha P = \alpha Q$

Semnul | va semnifica "alegere, alternativă cu, sau" : "alegere între x apoi P şi y apoi Q".

Exemple

X1 Posibilele mişcări ale markerului pe suprafaţa riglată sunt definite de procesul



$(sus \to STOP | dreapta \to dreapta \to sus \to STOP)$ □

X2 Un automat de schimbat monezi cu două posibilităţi de schimb pentru cinci penny (a se compara cu 1.1.2 X3 şi X4, unde nu aveam posibilităţi de alegere)

$SCH5C = in5p \to (ex1p \to ex1p \to ex1p \to ex2p \to SCH5C)$
$\quad | ex2p \to ex1p \to ex2p \to SCH5C)$ □

X3 Un automat care poate da ciocolată sau bomboane

$AVCB = \mu X.mon \to (choc \to X | bonbon \to X)$ □

X4 Un automat mai complex, care permite introducerea mai multor feluri de monezi şi furnizează atât napolitane mari şi mici cât şi restul aferent

$AVC = (in2p \to (mare \to AVC$
$\quad | mic \to ex1p \to AVC)$
$| in1p \to (mic \to AVC$
$\quad | in1p \to (mare \to AVC$
$\quad\quad | in1p \to STOP)))$

Deoarece are o funcţionare mai complicată, existând riscul de a introduce monezi şi a nu obţine nimic, automatul ar trebui însoţit de observaţii de forma

"ATENŢIE : Nu introduceţi trei monezi de un penny la rând." □

X5 Un automat care întâi dă ciocolata, având încredere că apoi va fi plătită. Secvenţa normală plată–ciocolată este de asemenea implementată

$$AUMCCM=\mu X.(mon\rightarrow choc\rightarrow X)$$
$$|choc\rightarrow mon\rightarrow X)$$ □

X6 Pentru a preveni pierderile putem impune automatului o plată iniţială, profitând de *AUMCCM*

$$AUS2=(mon\rightarrow AUMCCM)$$

Acest automat va permite introducerea a până la două monezi consecutive înainte de a furniza două ciocolate una după alta. În schimb, nu va elibera mai multe ciocolate decât monezi introduse. □

X7 Un proces de copiere se angajează în următoarele evenimente

pri.0 primeşte un zero pe canalul său de intrare
pri.1 primeşte un unu pe canalul său de intrare
emi.0 emite un zero pe canalul său de ieşire
emi.1 emite un unu pe canalul său de ieşire

Comportarea sa constă din repetarea unor perechi de evenimente. În fiecare ciclu, primeşte un bit pe care îl emite la ieşire

$$COPIEBIT=\mu X.(pri.0\rightarrow emi.0\rightarrow X$$
$$|pri.1\rightarrow emi.1\rightarrow X)$$

După cum se observă, procesul permite mediului de a face alegerea valorii care se primeşte, dar nu şi a celei care se copie. Aceasta comportare se deosebeşte de tratarea comunicaţiei din cap. 4. □

Definirea alegerii se poate uşor extinde la mai mult de două posibilităţi

$$(x\rightarrow P|y\rightarrow Q|\ \dots\ |z\rightarrow R)$$

Trebuie remarcat că operatorul | pentru alegere *nu* se aplică proceselor, fiind incorect sintactic o expresie de forma $P|Q$, unde P şi Q sunt procese. Motivul acestei constrângeri este eliminarea expresiilor de forma

$$(x\rightarrow P)|(x\rightarrow Q)$$

care numai în aparență oferă o alegere. Această expresie își va găsi rostul odată cu introducerea nedeterminismului, în paragraful 3.3. Pentru contextul actual, dacă x, y, z sunt evenimente diferite

$$(x \to P | y \to Q | z \to R)$$

poate fi privit ca un singur operator cu trei argumente, P, Q, R. Menționăm ca incorect sintactic și o expresie de forma

$$(x \to P | (y \to Q | z \to R))$$

În general, dacă B este o mulțime de evenimente și $P(x)$ este o expresie definind un proces pentru fiecare x diferit din B, atunci

$$(x{:}B \to P(x))$$

definește un proces care întâi oferă șansa oricărui eveniment y din B și apoi se comportă ca P. Se poate spune "x din B apoi P de x". În această construcție x este o variabilă locală, deci

$$(x{:}B \to P(x)) = (y{:}B \to P(y))$$

Vom spune că mulțimea B definește *meniul* inițial al procesului, deoarece include acțiuni între care trebuie făcută o alegere la începutul observării comportării.

Exemplu

X8 Un proces care se angajează oricând, în orice eveniment din alfabetul său notat A

$$\alpha RUN_A = A$$
$$RUN_A = (x{:}A \to RUN_A)$$ □

În cazul special al mulțimii B formate numai din evenimentul e, avem

$$(x{:}\{e\} \to P(x)) = (e \to P(e))$$

deoarece e este singurul eveniment inițial posibil. În cazul extrem când meniul inițial este mulțime vidă, nu se poate întâmpla nimic

$$(x{:}\{\} \to P(x)) = (y{:}\{\} \to P(y)) = STOP$$

Operatorul de alegere poate fi acum redefinit, folosind meniul, astfel

$$(a \rightarrow P | b \rightarrow Q) = (x{:}B \rightarrow R(x))$$

unde $B=\{a,b\}$ şi $R(x)=$ **if** $x=a$ **then** P **else** Q

O alegere între trei sau mai multe evenimente se poate exprima similar. Alegerea, prefixul şi *STOP* sunt de fapt cazuri speciale ale alegerii pe bază de meniu. Această observaţie va fi de real folos în formularea legilor generale ale proceselor (paragraful 1.3), precum şi la implementarea lor (paragraful 1.4).

1.1.4 Recursivitatea mutuală

Recursivitatea cum a fost introdusă permite definirea unui singur proces ca soluţie a unei singure ecuaţii. Metoda poate fi generalizată la un sistem de mai multe ecuaţii cu mai multe necunoscute. Pentru aceasta, trebuie ca toţi membrii drepţi ai ecuaţiilor să fie cu gardă iar necunoscutele (nume de procese) trebuie să apară o singură dată în câte un membru stâng al unei ecuaţii.

Exemplu

X1 Un automat de băuturi are două butoane etichetate ANANAS şi KIWI. Acţiunile de apăsare a butoanelor se noteză cu *setananas* şi *setkiwi*, iar acţiunile de eliberare a băuturilor cu *ananas* şi *kiwi*. Alegerea unei băuturi se face numai prin apăsarea butonului corespunzător înainte de a o obţine. Iată cum arată alfabetul şi ecuaţiile definind comportarea procesului AK. Sint utilizate două procese auxiliare A şi K, definite recursiv

$$\alpha AK = \alpha A = \alpha K = \{setananas,\ setkiwi,\ ananas,\ kiwi\}$$
$$AK = (setananas \rightarrow A \,|\, setkiwi \rightarrow K)$$
$$A = (ananas \rightarrow A \,|\, setkiwi \rightarrow K \,|\, setananas \rightarrow A)$$
$$K = (kiwi \rightarrow K \,|\, setananas \rightarrow A \,|\, setkiwi \rightarrow K)$$

După primul eveniment (apăsarea unui buton), automatul este în una din stările A sau K. În fiecare stare automatul poate servi băutura corespunzătoare sau poate fi comutat în cealaltă stare. Apăsarea butonului aceleiaşi stări este permisă şi are efectul de a menţine starea. □

Pentru sisteme mari de ecuaţii este posibilă folosirea variabilelor indexate

Exemplu

X2 Un obiect porneşte de pe sol şi poate face o mişcare în *sus*. Ulterior poate efectua mişcări în *sus* sau *jos*, cu excepţia cazului când se află pe sol şi nu poate face decât mişcarea *sus*. Totuşi când este pe sol , obiectul mai poate efectua şi mişcarea *pe_loc*. Fie n un număr natural, deci din {0,1,2,...}. Notăm cu MM_n, (marker mobil), comportarea obiectului când se află la n mişcări depărtare de sol. Comportarea sa iniţială este descrisă de

$$MM_0=(sus \to MM_1 | pe_loc \to MM_0)$$

iar restul ecuaţiilor, în număr infinit, sunt

$$MM_{n+1}=(sus \to MM_{n+2} | jos \to MM_n)$$

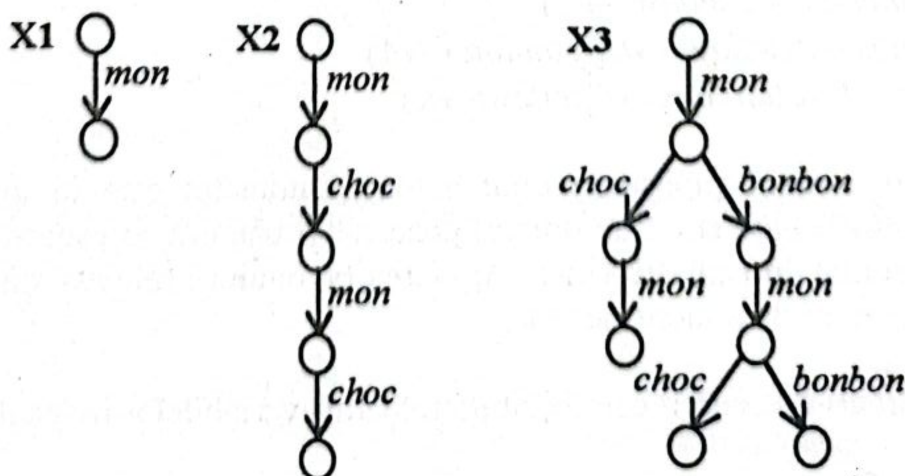
unde n este număr natural, după cum s-a specificat.

Cu toate că ecuaţia generală cuprinde pe MM_n, MM_{n+1}, MM_{n+2}, nu se poate folosi vreo metodă inductivă pentru a afla pe MM_n, deoarece el apare în ambii membri. Tot ceea ce se poate spune este că avem de-a face cu un sistem infinit de ecuaţii cu necunoscutele definite mutual recursiv, membrii drepţi ai ecuaţiilor fiind cu gardă, deci valizi din punctul nostru de vedere. □

1.2 Reprezentări grafice

Este util uneori de a reprezenta grafic un proces ca o structură arborescentă, formată din noduri şi arce. În terminologia curentă a automatelor, nodurile reprezintă stări iar arcele tranziţii. Există un nod rădăcină, starea iniţială, iar fiecare arc va fi etichetat cu evenimentul care este angajat în respectiva tranziţie. Arcele ce pornesc din acelaşi nod trebuie să aibă etichete diferite.

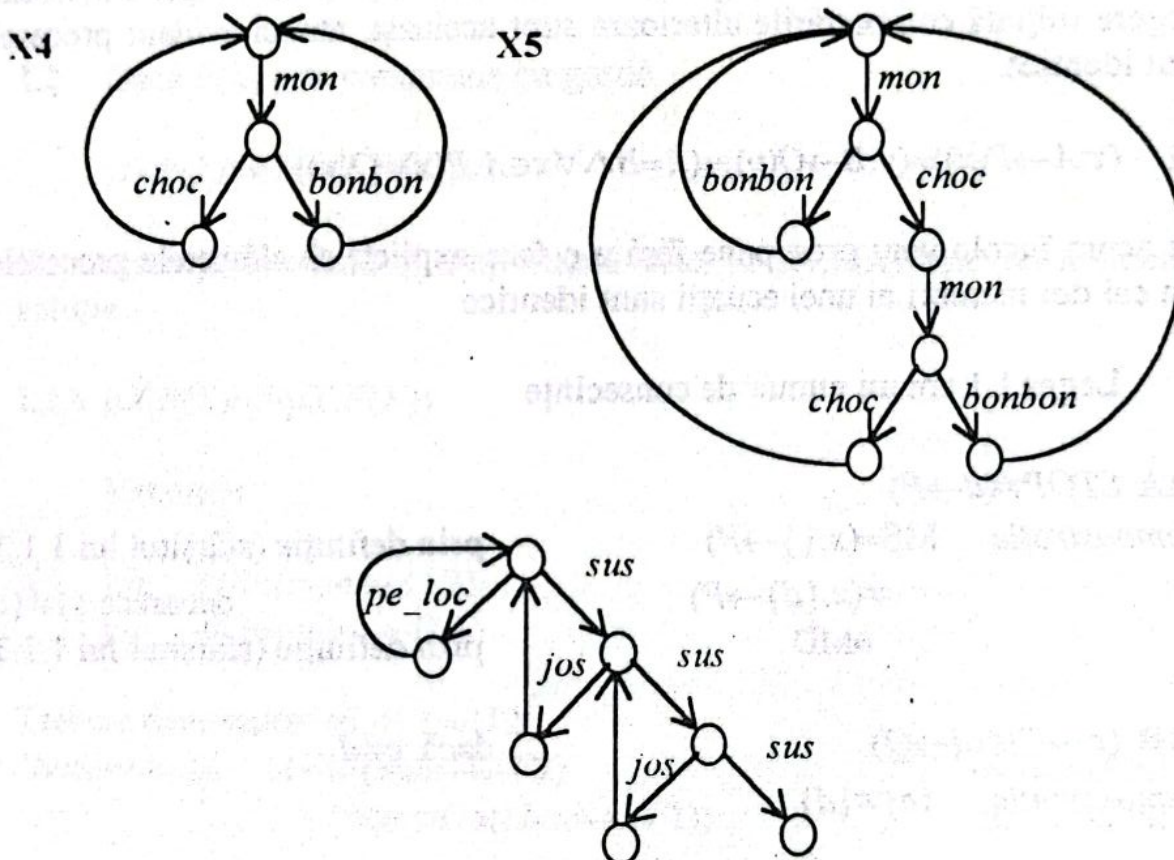
Exemple (1.1.1 X1,X2; 1.1.3 X3)



□

În aceste trei exemple starea finală este *STOP*, reprezentată de nodurile frunză. Pentru reprezentarea proceselor cu o comportare infinită trebuie introdusă convenţia existenţei unui arc neetichetat de la nodurile frunză la noduri strămoşi. Tranziţiile neetichetate se consideră a se produce imperceptibil şi imediat, ajungîndu-se în nodurile din care pleacă respectivele arce. □

De fapt, cele două reprezentări ilustrează acelaşi proces (1.1.3 X3), fiind clară slăbiciunea reprezentărilor grafice de a demonstra egalitatea proceselor. O altă slăbiciune a reprezentărilor grafice este în cazul proceselor cu un număr infinit de stări, de exemplu MM_0.



1.3 Legi

Chiar în condiţiile numărului mic de notaţii introduse până în prezent sunt destule posibilităţi de a descrie aceeaşi comportare. De exemplu, nu contează ordinea prezentării evenimentelor într-o alegere

$$(x \to P \mid y \to Q) = (y \to Q \mid x \to P)$$

De asemenea este evident că un proces care nu mai poate avea nici o comportare este diferit de unul ce se angajează în cel puţin un eveniment

$(x \to P) \neq STOP$

Pentru a înţelege corect notaţiile şi a opera cu ele, trebuie să învăţăm a recunoaşte care expresii descriu acelaşi obiect şi care nu, la fel cum toţi ştim că expresia $(x+y)$ este tot una cu $(y+x)$. Identitatea sau nu a proceselor cu acelaşi alfabet poate fi dovedită prin apelarea la legi algebrice asemănătoare celor aritmetice.

Prima dintre ele (L1 de mai jos) se ocupă de operatorul alegere. Ea asigură că două procese descrise cu ajutorul alegerii sunt diferite dacă oferă şanse diferite în primul pas sau dacă după acelaşi prim pas, ele se comportă diferit. Dacă totuşi alegerile iniţiale sunt aceleaşi şi dacă pentru fiecare alegere iniţială comportările ulterioare sunt aceleaşi, atunci evident procesele sunt identice.

L1 $(x{:}A \to P(x)) = (y{:}B \to Q(y)) \equiv (A = B \land \forall x \in A . P(x) = Q(y))$

De acum încolo vom presupune fără a o face explicit că alfabetele proceselor din cei doi membri ai unei ecuaţii sunt identice

Legea L1 are un număr de consecinţe

L1A $STOP \neq (d \to P)$

Demonstraţie MS$=(x{:}\{\} \to P)$ prin definiţie (sfârşitul lui 1.1.3)

$\neq (x{:}\{d\} \to P)$ deoarece $\{\} \neq \{d\}$

=MD prin definiţie (sfârşitul lui 1.1.3)

L1B $(c \to P) \neq (d \to Q)$ dacă $c \neq d$

Demonstraţie $\{c\} \neq \{d\}$

L1C $(c \to P \mid d \to Q) = (d \to Q \mid c \to P)$

Demonstraţie Definim $R(x) = P$ dacă $x = c$

$= Q$ dacă $x = d$

MS$=(x{:}\{c,d\} \to R(x))$ prin definiţie

$=(x{:}\{d,c\} \to R(x))$ deoarece $\{c,d\} = \{d,c\}$

=MD prin definiţie

L1D $(c \to P) = (c \to Q) \equiv P = Q$

Demonstraţie $\{c\} = \{c\}$

Aceste legi permit demonstrarea unor teoreme simple.

Exemple

X1 $(mon \rightarrow choc \rightarrow mon \rightarrow choc \rightarrow STOP) \neq (mon \rightarrow STOP)$

Demonstrație: din L1D și L1A □

X2 $\mu X.(mon \rightarrow (choc \rightarrow X | bonbon \rightarrow X)) = \mu X.(mon \rightarrow (bonbon \rightarrow X | choc \rightarrow X))$

Demonstrație: din L1C □

Pentru a putea demonstra teoreme mai generale privind recursivitatea proceselor este necesar de a introduce o lege care stabilește existența unei soluții unice în cazul ecuațiilor recursive cu gardă efectivă.

L2 Dacă $F(X)$ este o expresie cu gardă

$$(Y=F(Y)) \equiv (Y=\mu X.F(X))$$

O consecință imediată și importantă este că $\mu X.F(X)$ este de asemenea o soluție

L2A $\mu X.F(X)=F(\mu X.F(X))$

Exemplu

X3 Fie $AV1=(mon \rightarrow AV2)$
și $AV2=(choc \rightarrow AV1)$

Trebuie demonstrat că $AV1=AVS$

Demonstrație $AV1=(mon \rightarrow AV2)$ definiția lui $AV1$
$=(mon \rightarrow (choc \rightarrow AV1))$ definiția lui $AV2$

Deci $AV1$ este și ea o soluție a ecuației recursive descrisă de AVS. Deoarece ecuația este cu gardă efectivă are o singură soluție, $AV1$ și AVS fiind două nume diferite pentru aceeași soluție.

Această teoremă apare ca evident adevărată. Singurul scop al demonstrației este de a arăta printr-un exemplu că legile sunt suficient de puternice pentru a stabili fapte de acest gen. Când se demonstrază fapte evidente din legi mai puțin evidente este important de a justifica fiecare linie din demonstrație, ca o verificare că demonstrația nu folosește concluziile drept ipoteze.

□

Legea L2 poate fi extinsă la recursivitatea mutuală. Un sistem de ecuații recursive mutual poate fi scris în formă generală folosind indici

$X_i = F(i,X)$ pentru toţi $i \in S$

unde S este mulţimea indecşilor ecuaţiilor, cu un singur index per ecuaţie
şi X este un masiv de procese cu indici în domeniul mulţimii S
şi $F(i,X)$ este o expresie cu gardă efectivă.

În aceste condiţii legea L3 precizează că există un singur masiv de procese X, ale cărui elemente satisfac toate ecuaţiile

L3 În condiţiile de mai sus

dacă $(\forall i{:}S.(X_i = F(i,X) \wedge Y_i = F(i,Y)))$ atunci $X = Y$

1.4 Implementarea proceselor

În condiţiile notaţiilor introduse până acum orice proces P poate fi scris sub forma

$(x{:}B \rightarrow F(x))$

unde F este o funcţie definită pe mulţimea simbolurilor proceselor, cu valori în mulţimea proceselor şi B poate fi vidă (în cazul lui *STOP*), poate conţine un singur element (în cazul prefixului), sau poate conţine mai multe elemente (în cazul alegerii). Pentru procesele definite recursiv am insistat că recursivitatea trebuie să fie cu gardă efectivă astfel că poate fi scrisă

$\mu X.(x{:}B \rightarrow F(x,X))$

iar desfăşurat, ia următoarea formă folosind L2A

$(x{:}B \rightarrow F(x, \mu X(x{:}B \rightarrow F(x,X))))$

Având în vedere cele arătate, un proces poate fi privit ca o *funcţie* F, cu domeniul B, format dintr-o mulţime de evenimente în care procesul se poate angaja iniţial. Pentru fiecare eveniment x din B, $F(x)$ defineşte comportarea ulterioară a procesului după angajarea în evenimentul x.

Aceste observaţii permit reprezentarea unui proces ca o funcţie într-un limbaj de programare funcţional precum LISP-ul. Fiecare eveniment din alfabetul procesului se va reprezenta ca un atom, de exemplu "*MON*, "*BONBON*. Un proces devine o funcţie cu argumente astfel de simboluri. Dacă simbolul *nu* este un prim eveniment posibil pentru proces, funcţia returnează

un simbol special numit "*BLIP*, folosit numai în acest scop. De exemplu, deoarece *STOP* nu se angajează în nici un eveniment, poate fi definit

$STOP = \lambda x.\text{"}BLIP$

Dacă însă argumentul este un eveniment posibil pentru proces se returnează ca rezultat altă funcţie, şi anume comportarea ulterioară a procesului. Astfel

($mon \to STOP$) se reprezintă ca o funcţie

$\lambda x.$ **if** $x=\text{"}MON$ **then** $STOP$
else "*BLIP*

Acest ultim exemplu ne arată facilitatea LISP-ului de a returna o funcţie (*STOP*) ca rezultat al unei alte funcţii. LISP-ul permite de asemenea transmiterea unei funcţii ca argument al unei alte funcţii, facilitate folosită în reprezentarea operatorului prefix ($c \to P$)

$prefix(c,P) = \lambda x.$ **if** $x=c$ **then** P
else "*BLIP*

O funcţie care să reprezinte alegerea generală binară ($c \to P \mid d \to Q$) necesită patru parametri

$alegere2(c,P,d,Q) = \lambda x.$ **if** $x=c$ **then** P
else if $x=d$ **then** Q
else "*BLIP*

Procesele definite recursiv pot fi reprezentate cu ajutorul facilităţii *LABEL* din LISP. De exemplu, automatul simplu de vânzare ($\mu X.mon \to choc \to X$) se reprezintă ca

LABEL X.prefix("*MON*,*prefix*("*CHOC*,*X*))

LABEL poate fi folosit pentru a implementa şi recursivitatea mutuală. De exemplu, MM_n (1.1.4 X2) poate fi considerată ca o funcţie pe mulţimea numerelor naturale şi având codomeniu procese (la rândul lor funcţii). Astfel MM_n poate fi definită

$MM = LABEL\ X.\ \lambda n.$
if $n=0$ **then** *alegere2*("*LATERAL*,$X(0)$,"*SUS*,$X(1)$)
else *alegere2*("*SUS*,$X(n+1)$,"*JOS*,$X(n-1)$)

Procesul iniţial este deci $MM(0)$.

Procesul inițial este deci *MM*(0).

Dacă *P* este o funcție reprezentând un proces iar *A* lista ce conține simbolurile din alfabetul său, funcția LISP

meniu(*A*,*P*)

returnează lista tuturor simbolurilor din *A* ce pot fi prime evenimente în existența lui *P*

meniu(*A*,*P*)= **if** *A*=*NIL* atunci *NIL*
then if *P*(*car*(*A*))="*BLIP* **then** *meniu*(*cdr*(*A*),*P*)
else *cons*(*car*(*A*),*meniu*(*cdr*(*A*),*P*))

Dacă *x* este în *meniu*(*A*,*P*), *P*(*x*) nu poate fi "*BLIP* și de aceea este o funcție definind comportarea ulterioară a lui *P* după angajarea în evenimentul *x*. Dacă *y* este în *meniu*(*A*,*P*(*x*)), atunci *P*(*x*)(*y*) dă comportarea după producerea lui *x* și *y*. Se sugerează în acest mod o metodă eficientă de a explora comportarea procesului. Se poate scrie un program care întâi afișează valoarea lui *meniu*(*A*,*P*) iar după aceea preia un simbol de la tastatură. Dacă simbolul nu face parte din meniu, se va genera un semnal de eroare și va fi ignorat. Dacă simbolul face parte, el este acceptat iar bucla este reluată prin înlocuirea lui *P* cu rezultatul aplicării simbolului respectiv lui *P*. Programul poate fi terminat prin introducerea simbolului "*END*. Dacă *k* este lungimea secvenței de simboluri preluate la intrare, funcția ce returnează secvența de ieșiri cerută este

interact(*A*,*P*,*k*)=*cons*(*meniu*(*A*,*P*),
if *car*(*k*)="*END* **then** *NIL*
else if *P*(*car*(*k*))="*BLIP* **then**
cons("*BLIP*,*interact*(*A*,*P*,*cdr*(*k*)))
else *interact*(*A*,*P*(*car*(*k*)),*cdr*(*k*)))

Notațiile folosite mai sus pentru definirea funcțiilor LISP sunt generale și este nevoie de o adaptare a S-expresiilor pentru o implementare particulară de LISP. De exemplu, în LISPkit funcția prefix va avea forma

```
(prefix
    lambda
    (a p)
    (lambda (x) (if (eq x a) p (quote BLIP))))
```

Din fericire, în carte vom folosi o mică parte din LISP-ul pur funcţional, astfel că nu vor fi probleme mari cu translatarea şi executarea proceselor într-un dialect de pe un anumit calculator.

Dacă aveţi la dispoziţie mai multe versiuni de LISP, alegeţi una cu o corectă legare statică a variabilelor. Un LISP cu o evaluare amânată este de asemenea convenabil din cauza transcrierii directe a ecuaţiilor recursive, fără folosirea lui *LABEL*

AVS=prefix("MON,prefix("CHOC,AVS))

Dacă intrarea şi ieşirea sunt implementate prin evaluare progresivă, al treilea paramentru al funcţiei *interact* poate fi tastatura. Meniul pentru procesul *P* va fi primul parametru. Selectând şi introducând un simbol din meniurile succesive, un utilizator poate explora interactiv comportarea procesului *P*. În alte versiuni de LISP, funcţia *interact* trebuie rescrisă pentru a realiza un efect similar al intrărilor şi ieşirilor. Se poate spune că un proces se execută reprezentat fiind printr-o funcţie LISP, deci s-a realizat o *implementare* a procesului. La fel, funcţia LISP corespunzătoare lui *prefix* operează cu aceste reprezentări, fiind o implementare a operatorului corespunzător dintre procese.

1.5 Urme

O *urmă* a comportării unui proces este o secvenţă finită de simboluri, înregistrând evenimentele în care procesul s-a angajat până la un anumit moment dat de timp. Ne putem imagina că există un observator cu un carneţel, care urmăreşte procesele şi notează numele fiecărui eveniment ce apare. Cum am presupus deja, ignorăm posibilitatea apariţiei simultane a evenimentelor. Dacă totuşi există şi astfel de evenimente, observatorul le va nota într-o ordine oarecare ce nu are importanţă.

O urmă va fi definită ca o secvenţă de simboluri, separate prin virgulă şi incluse între < >

<*x,y*> constă din două evenimente, *x* urmat de *y*.

<*x*> o secvenţă ce conţine un singur eveniment *x*.

<> secvenţă vidă, fără nici un eveniment.

Exemple

X1 O urmă a automatului simplu de vânzare *AVS* (1.1.2 X2) după ce a servit primii doi clienţi

□

<mon,choc,mon,choc>

X2 O urmă a aceluiaşi automat înainte ca al doilea client să-şi extragă ciocolata sa

<mon,choc,mon>

Trebuie menţionat că atât procesul cât şi observatorul nu au noţiunea de tranzacţie completă. Nerăbdarea sau foamea clientului, precum şi posibilitatea automatului de a furniza produse sunt evenimente ce nu figurează în alfabetul proceselor, deci nu pot fi înregistrate, observate. □

X3 Înainte ca un proces să se angajeze în vreun eveniment, carneţelul observatorului este gol. Acest lucru se poate reprezenta prin urma vidă

<>

Orice proces porneşte de fapt cu această urmă. □

X4 Automatul de vânzare perfecţionat, *AVC* (1.1.3 X4) are următoarele şapte urme de lungime cel mult doi

<>

<in2p> *<in1p>*

<in2p,mare> *<in2p,mic>* *<inp,in1p><in1p,mic>*

Evident numai una din cele patru urme de lungime doi apare odată. Alegerea între ele este făcută prin intermediul intenţiilor clientului. □

X5 Urma unui *AVC* dacă se ignoră avertismentul introducerii consecutive a trei monezi este

<in1p,in1p,in1p>

Urma nu poate semnala defectarea automatului. Defectarea poate fi detectată prin aceea că, printre atâtea urme posibile nu există nici una de forma

<in1p,in1p,in1p,x>

pentru orice eveniment *x*. □

1.6 Operaţii cu urme

Urmele joacă un rol principal în înregistrarea, descrierea şi înţelegerea comportamentului proceselor. În acest paragraf vom explora câteva din proprietăţile generale ale urmelor şi operaţiile cu ele. Vom folosi următoarele convenţii

s, t, u	pentru urme
S, T, U	pentru mulţimi de urme
f, g, h	pentru funcţii

1.6.1 Concatenarea

Cea mai importantă operaţie cu urme este concatenarea, care construieşte o urmă din doi operanzi *s* şi *t*, în această ordine. Rezultatul se va nota

$$s\hat{}t$$

De exemplu

$$\langle mon,choc\rangle\hat{}\langle mon,bonbon\rangle=\langle mon,choc,mon,bonbon\rangle$$
$$\langle in1p\rangle\hat{}\langle in1p\rangle=\langle in1p,in1p\rangle$$
$$\langle in1p,in1p\rangle\hat{}\langle\rangle=\langle in1p,in1p\rangle$$

Cea mai importantă proprietate a concatenării este asociativitatea şi faptul că secvenţa $\langle\rangle$ este element unitate

L1 $s\hat{}\langle\rangle=\langle\rangle\hat{}s=s$
L2 $s\hat{}(t\hat{}u)=(s\hat{}t)\hat{}u$

Următoarele legi sunt atât evidente cât şi utile

L3 $s\hat{}t=s\hat{}u\equiv t=u$
L4 $s\hat{}t=u\hat{}t\equiv s=u$
L5 $s\hat{}t=\langle\rangle\equiv s=\langle\rangle\wedge t=\langle\rangle$

Fie *f* o funcţie cu domeniul şi codomeniul urme. Funcţia se spune că este *strictă* dacă este identitate pentru urma vidă

$$f(\langle\rangle)=\langle\rangle$$

Se spune că *f* este *distributivă* faţă de concatenare dacă

$f(s\hat{}t)=f(s)\hat{}f(t)$

Toate funcţiile distributive sunt stricte.

Dacă n este un număr natural, definim t^n ca fiind n copii concatenate ale lui t. Prin inducţie după n avem

L6 $t^0=\langle\rangle$
L7 $t^{n+1}=t\hat{}t^n$

Definiţia generează deci cele două legi. Se mai pot deduce două ce pot fi dovedite uşor

L8 $t^{n+1}=t^n\hat{}t$
L9 $(s\hat{}t)^{n+1}=s\hat{}(t\hat{}s)^n\hat{}t$

1.6.2 Restricţia

Expresia $(t\upharpoonright A)$ semnifică urma t restricţionată la simbolurile din mulţimea A. Ea se formează simplu din t omiţând simbolurile ce nu fac parte din A. De exemplu

$\langle pe_loc, sus, jos, pe_loc\rangle\upharpoonright\{sus, jos\}=\langle sus, jos\rangle$

Restricţia este distributivă şi de aceea strictă

L1 $\langle\rangle\upharpoonright A=\langle\rangle$
L2 $(s\hat{}t)\upharpoonright A=(s\upharpoonright A)\hat{}(t\upharpoonright A)$

Efectul asupra secvenţei cu un singur element este evident

L3 $\langle x\rangle\upharpoonright A=\langle x\rangle$ dacă $x\in A$
L4 $\langle y\rangle\upharpoonright A=\langle\rangle$ dacă $y\notin A$

O funcţie distributivă este definită unic prin efectul asupra secvenţei cu un singur element, deoarece efectele asupra secvenţelor de lungimi mai mari pot fi calculate prin aplicarea distributivităţii fiecărui element şi concatenarea rezultatelor. De exemplu, dacă $y\neq x$

$\langle x,y,x\rangle\upharpoonright\{x\}=(\langle x\rangle\hat{}\langle y\rangle\hat{}\langle x\rangle)\upharpoonright\{x\}$
$=(\langle x\rangle\upharpoonright\{x\})\hat{}(\langle y\rangle\upharpoonright\{x\})\hat{}(\langle x\rangle\upharpoonright\{x\})$ din L2
$=\langle x\rangle\ \hat{}\langle\rangle\ \hat{}\langle x\rangle$ din L3, L4
$=\langle x,x\rangle$

Următoarele legi ne arată relaţia dintre restricţie şi operaţiile cu mulţimi. O urmă restricţionată la o mulţime vidă ne dă o secvenţă vidă. O restricţie succesivă la două mulţimi are acelaşi efect ca şi restricţia la intersecţia mulţimilor. Aceste două legi pot fi demonstrate prin inducţie după lungimea secvenţelor.

L5 $s\restriction\{\ \}=\langle\rangle$
L6 $(s\restriction A)\restriction B=s\restriction(A\cap B)$

1.6.3 Cap şi coadă

Dacă s este o secvenţă nevidă, primul ei simbol se va nota s_0, iar rezultatul eliminării primului simbol se va nota s'. De exemplu

$\langle x,y,x\rangle_0=x$.
$\langle x,y,x\rangle'=\langle y,x\rangle$

Ambele operaţii sunt inoperante pe secvenţa vidă

L1 $(\langle x\rangle\frown s)_0=x$
L2 $(\langle x\rangle\frown s)'=s$
L3 $s=(\langle s_0\rangle\frown s')$ dacă $s\neq\langle\rangle$

Următoarea lege dă o metodă comodă de a arăta că două urme sunt identice

L4 $s=t\equiv(s=t=\langle\rangle\vee(s_0=t_0\wedge s'=t'))$

1.6.4 Stea

Mulţimea A^* este mulţimea tuturor secvenţelor finite (inclusiv $\langle\rangle$) ce se formează din simbolurile mulţimii A. Când astfel de urme sunt restricţionate la mulţimea A, ele rămân nemodificate. Această proprietate permite o definiţie simplă

$A^*=\{s|s\restriction A=s\}$.

Următoarele legi sunt consecinţe ale definiţiei

L1 $\langle\rangle\in A^*$
L2 $\langle x\rangle\in A^*\equiv x\in A$
L3 $(s\frown t)\in A^*\equiv s\in A^*\wedge t\in A^*$

Ele sunt suficient de puternice pentru a determina dacă o urmă este sau nu un membru al lui A^*. De exemplu, dacă $x \in A$ şi $y \notin A$

$$\langle x,y\rangle \in A^* \equiv (\langle x\rangle \frown \langle y\rangle) \in A^*$$
$$\equiv (\langle x\rangle \in A^*) \wedge (\langle y\rangle \in A^*) \quad \text{din L3}$$
$$\equiv true \wedge false \quad \text{din L2}$$
$$\equiv false$$

Altă lege utilă ar putea fi definirea recursivă a lui A^*

L4 $A^* = \{t \mid t = \langle\rangle \vee (t_0 \in A \wedge t' \in A^*\}$.

1.6.5 Ordonarea

Dacă s este o subsecvenţă iniţială a lui t, este posibil de a găsi o extensie u a lui s astfel ca $s \frown u = t$. De aceea definim o relaţie de ordine

$$s \le t = (\exists u . s \frown u = t)$$

şi spunem că s este un *prefix* a lui t. De exemplu

$$\langle x,y\rangle \le \langle x,y,x,w\rangle$$
$$\langle x,y\rangle \le \langle z,y,x\rangle \equiv x = z$$

Relaţia $\le$ este o ordine parţială, cu cel mai mic element $\langle\rangle$, cum rezultă din legile L1-L4

L1 $\langle\rangle \le s$ cel mai mic element
L2 $s \le s$ reflexivitate
L3 $s \le t \wedge t \le s \Rightarrow s = t$ antisimetrie
L4 $s \le t \wedge t \le u \Rightarrow s \le u$ tranzitivitate

Următoarea lege împreună cu L1 face posibil verificarea ordinii $s \le t$

L5 $(\langle x\rangle \frown s) \le t \equiv t \ne \langle\rangle \wedge x = t_0 \wedge s \le t'$

Prefixele unei secvenţe date sunt total ordonate

L6 $s \le u \wedge t \le u \Rightarrow s \le t \vee t \le s$

Dacă s este o subsecvenţă a lui t (nu-i necesar iniţială), spunem că avem relaţia s **in** t. Aceasta se poate defini astfel

$s \textbf{ in } t = (\exists u,v.\ t = u \frown s \frown v)$

Această relaţie este de asemenea o ordine parţială, deoarece satisface legile L1-L4 de mai sus. Relaţia mai satisface şi legea

L7 $(\langle x \rangle \frown s) \textbf{ in } t \equiv t \neq \langle\rangle \wedge (t_0 = x \wedge s \leq t') \vee (\langle x \rangle \frown s) \textbf{ in } t')$

O funcţie f cu domeniul şi codomeniul mulţimi de urme se spune că este *monotonă* dacă respectă ordinea $\leq$

$f(s) \leq f(t)$ dacă $s \leq t$

Toate funcţiile distributive sunt monotone, de exemplu

L8 $s \leq t \Rightarrow (s \upharpoonright A) \leq (s \upharpoonright A)$

O funcţie de două argumente poate fi monotonă în oricare din ele separat, celălalt fiind ţinut constant. De exemplu, concatenarea este monotonă în al doilea argument (dar nu în primul)

L9 $t \leq u \Rightarrow (s \frown t) \leq (s \frown u)$

O funcţie monotonă în ambele argumente se numeşte pur şi simplu monotonă.

1.6.6 Lungime

Lungimea unei urme t este notată $\#t$. De exemplu

$\#\langle x,y,x \rangle = 3$

Legile care definesc # sunt

L1 $\#\langle\rangle = 0$
L2 $\#\langle x \rangle = 1$
L3 $\#\langle s \frown t \rangle = (\#s) + (\#t)$

Numărul apariţiilor simbolilor din A în secvenţa t este dată de $\#(t \upharpoonright A)$.

L4 $\#(t \upharpoonright (A \cup B)) = \#(t \upharpoonright A) + \#(t \upharpoonright B) - \#(t \upharpoonright (A \cap B))$
L5 $s \leq t \Rightarrow \#s \leq \#t$
L6 $\#(t^n) = n \times (\#t)$

Numărul aparițiilor simbolului x în urma s este definită de

$$s\downarrow x=\#(s\upharpoonright\{x\})$$

1.7 Implementarea urmelor

În vederea implementării urmelor și a operațiilor cu ele, folosim limbajul de nivel înalt pentru prelucrarea listelor, foarte potrivit pentru aceasta, LISP. Urmele vor fi reprezentate prin liste de atomi corespunzători evenimentelor

$\langle\rangle=NIL$
$\langle mon\rangle=cons(\text{"}MON,NIL)$
$\langle mon,choc\rangle=\text{"}(MON\ \ CHOC)$
adică $cons(\text{"}MON,cons(\text{"}CHOC,NIL))$

Operațiile cu urme pot fi ușor implementate cu ajutorul funcțiilor de liste

$t_0=car(t)$
$t'=cdr(t)$
$\langle x\rangle\char`^s=cons(x,s)$

Concatenarea este implementată cu ajutorul binecunoscutei funcții *append*, definită recursiv

$s\char`^t=append(s,t)$

unde $append(s,t)=$ **if** $s=NIL$ **then** t **else** $cons(car(s),append(cdr(s),t))$

corectitudinea definiției decurge din legile

$\langle\rangle\char`^t=t$
$s\char`^t=\langle s_0\rangle\char`^(s'\char`^t)$ dacă $s\neq\langle\rangle$

Terminarea funcției LISP *append* este garantată de faptul că lista din primul argument al fiecărui apel recursiv este mai scurtă decât în precedentul apel. Datorită unor raționamente analoage se poate stabili corectitudinea implementărilor altor operații definite mai jos.

Pentru a implementa restricția, folosim o listă B drept mulțimea elementelor. Testul $(x\in B)$ se realizează prin apelul funcției

$estemembru(x,B)=$ **if** $B=NIL$ **then** *false*

else if *x=car(B)* **then** *true*
else *estemembru(x,cdr(B))*

($s\restriction A$) poate fi implementat prin funcţia

restrict(s,B)= **if** *s=NIL* **then** *NIL*
else if *estemembru(car(s),B)*
then *cons(car(s),restrict(cdr(s),B))*
else *restrict(cdr(s),B)*

Testul (s^t) se implementează cu o funcţie care returnează *true* sau *false*. Ne bazăm pe 1.6.5 L1 şi L5

esteprefix(s,t)= **if** *s=NIL* **then** *true*
else if *t=NIL* **then** *false*
else *car(s)=car(t)*$\wedge$*esteprefix(cdr(s),cdr(t))*

1.8 Urmele proceselor

În paragraful 1.6 urma unui proces a fost introdusă ca fiind înregistrarea secvenţială a comportării procesului până la un moment dat. Înainte de începerea procesului nu se poate şti care din urmele sale posibile va fi urmată. Alegerea este independentă de controlul procesului, fiind determinată de factorii de mediu. Este clar că mulţimea tuturor urmelor posibile ale unui proces *P poate* fi cunoscută apriori, astfel că putem defini *urme(P)* pentru a o defini.

Exemple

X1 Singura urmă ce rezultă din comportarea procesului *STOP* este <>. Carneţelul observatorului în cazul acestui proces rămâne gol totdeauna

urme(STOP)={<>} □

X2 Există numai două urme ale automatului de vânzare în care se introduce o monedă şi apoi se defectează

urme(mon→STOP)={<>,<*mon*>} □

X3 Un ceas care ticăie mereu

urme(μX.*tic*→*X*)={<>,<*tic*>,<*tic,tic*>,...}
={*tic*}*

La cele mai multe din procese, mulţimea urmelor este infinită, cu toate că fiecare urmă în parte este finită. □

X4 Un automat simplu de vânzare

$$urme(\mu X.mon \rightarrow choc \rightarrow X)=\{s \mid \exists n.\ s \leq <mon,choc>^n\}$$ □

1.8.1 Legi

În acest paragraf vom deduce mulţimea urmelor unui proces dat folosind notaţiile introduse. Cum s-a arătat mai sus, *STOP* are o singură urmă

L1 $urme(STOP)=\{t \mid t=<>\}=\{<>\}$

Urma lui $(c \rightarrow P)$ poate fi vidă, deoarece $<>$ este urma comportării oricărui proces până la momentul când se angajează în prima acţiune. Astfel, orice urmă nevidă a lui P începe cu c şi coada este o posibilă urmă a procesului.

L2 $urme(c \rightarrow P)=\{t \mid t=<> \vee (t_0=c \wedge t' \in urme(P))\}=\{<>\} \cup \{<c>\^t \mid t \in urme(P)\}$

Urma comportării unui proces care oferă alegere iniţială între evenimente trebuie să fie urma uneia din alternative

L3 $urme(c \rightarrow P|d \rightarrow P)=\{t \mid t=<> \vee (t_0=c \wedge t' \in urme(P)) \vee (t_0=d \wedge t' \in urme(Q))\}$

Aceste trei legi pot fi sintetizate într-o singură lege, mai generală, despre alegere

L4 $urme(x:B \rightarrow P(x))=\{t \mid t=<> \vee (t_0 \in B \wedge t' \in urme(P(t_0)))\}$

Ceva mai complicat este de a afla mulţimea urmelor unui proces definit recursiv. Un astfel de proces este soluţia unei ecuaţii de forma

$$X=F(X)$$

Mai întâi, definim recursivitatea printr-o inducţie

$$F^0(X)=X$$
$$F^{n+1}(X)=F(F^n(X))$$
$$=F^n(F(X))$$

$$=\underbrace{F(...(F(F(X)))...)}_{n\ ori}$$

Apoi, dacă F este cu gardă efectivă, putem defini

L5 $urme(\mu X{:}A.F(X))=\bigcup_{n\geq 0} urme(F^n(STOP_A))$

Exemple

X1 Să ne reamintim cum am definit RUN_A în 1.1.3 X8

$\mu X{:}A.F(X)$

unde $F(X)=(x{:}A\rightarrow X)$

Dorim să demonstrăm că

$urme(RUN_A)=A^*$

Demonstrație. $A^*=\bigcup_{n\geq 0}\{s \mid s\in A^* \wedge \#s\leq n\}$

Folosind L5 este suficient de demonstrat pentru orice n că

$urme(F^n(STOP_A))=\{s \mid s\in A^* \wedge \#s\leq n\}$

Acest lucru îl vom face prin inducție după n

(1) pentru $n=0$

$$urme(STOP_A)=\{\langle\rangle\}$$
$$=\{s \mid s\in A^* \wedge \#s\leq 0\}$$

(2) $urme(F^{n+1}(STOP_A))$

$=urme(x{:}A\rightarrow F^n(STOP_A))$ definiția lui F, F^{n+1}

$=\{t \mid t=\langle\rangle \vee (t_0\in A \wedge t'\in urme(F^n(STOP_A)))\}$ L4

$=\{t \mid t=\langle\rangle \vee (t_0\in A \wedge (t'\in A^* \wedge \#t'\leq n))\}$ ipoteza de inducție

$=\{t \mid (t=\langle\rangle \vee (t_0\in A \wedge t'\in A^*)) \wedge \#t\leq n+1\}$ proprietatea lui #

$=\{t \mid t\in A^* \wedge \#t\leq n+1\}$ 1.6.4 L4 □

X2 Vrem să demonstrăm corectitudinea lui 1.8 X4, adică

$$urme(AVS)=\bigcup_{n\geq 0}\{s \mid s\leq \langle mon,choc\rangle^n\}$$

Demonstrație. Propoziția de inducție este

$$urme(F^n(AVS))=\{t \mid t\leq \langle mon,choc\rangle^n\}$$

unde $F(X)=(mon\rightarrow choc\rightarrow X)$

(1) $urme(STOP)=\{\langle\rangle\}=\{s \mid s\leq \langle mon,choc\rangle^n\}$ 1.6.1 L6

(2) $urme(mon\rightarrow choc\rightarrow F^n(STOP))$

$=\{\langle\rangle,\langle mon\rangle\}\cup\{\langle mon,choc\rangle^\wedge t \mid t\in urme(F^n(STOP_A))\}$ aplic L2 de două ori

$=\{\langle\rangle,\langle mon\rangle\}\cup\{\langle mon,choc\rangle^\wedge t \mid t\leq \langle mon,choc\rangle^n\}$ ipoteza de inducție

$=\{s \mid s=\langle\rangle\vee(s=\langle mon\rangle\vee\exists t.s=\langle mon,choc\rangle^\wedge t\wedge t\leq \langle mon,choc\rangle^n\}$

$=\{s \mid s\leq \langle mon,choc\rangle^{n+1}\}$

Concluzia decurge din L5 □

Cum s-a menționat în paragraful 1.5, o urmă este o secvență de simboluri înregistrând evenimentele în care s-a angajat procesul P până la un moment dat de timp. Astfel, rezultă clar că $\langle\rangle$ este urma oricărui proces până la momentul când se angajează în primul eveniment. Mai mult, dacă $s^\wedge t$ este urma unui proces până la un moment de timp, atunci s este urma până la un moment anterior. În fine, orice eveniment care apare trebuie să fie în alfabetul procesului considerat. Aceste trei concluzii sunt formalizate în legile următoare

L6 $\langle\rangle\in urme(P)$

L7 $s^\wedge t\in urme(P)\Rightarrow s\in urme(P)$

L7 $urme(P)\subseteq(\alpha P)^*$

Există o strânsă legătură între urmele unui proces și reprezentarea sub formă de arbore a comportării sale. Urma comportării procesului până la un moment de timp, corespunzător unui nod al arborelui, este secvența etichetelor evenimentelor pe calea de la rădăcină la acel nod. De exemplu, pentru arborele corespunzător lui *AVC*, fig. 1.1, urma obținută prin parcurgerea căii de la rădăcină la nodul punctat plin este

$\langle in2p,mic,ex1p\rangle$

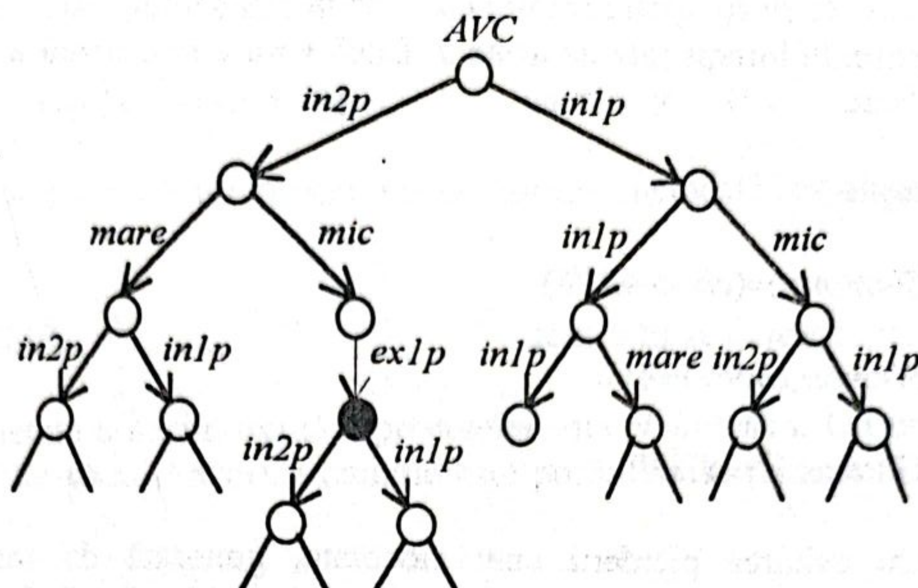


Figura 1.1

Căile sau subcăile din arbore sunt urme pentru procesul dat. Acest lucru este concluzionat în L7. Legea L6 semnifică o urmă vidă pentru calea de la nodul rădăcină la el însuşi. Ultima lege, L8, înseamnă că *urme*-le unui proces sunt incluse în mulţimea căilor ce pleacă din rădăcină.

Reciproc, deoarece arcele incidente exterior oricărui nod sunt etichetate cu evenimente diferite, orice urmă corespunde unei singure căi de la nodul rădăcină la un nod oarecare. Astfel, orice mulţime de urme satisfăcând L6 şi L7 este o reprezentare matematică convenabilă a unui arbore cu arcele incidente exterior fiecărui nod etichetate cu evenimente diferite.

1.8.2 Implementare

Presupunem că un proces este implementat cu ajutorul unei funcţii LISP *P* şi fie *s* o urmă. Putem testa dacă *s* este o posibilă urmă a lui *P* cu funcţia

esteurmă(*s*,*P*)= **if** *s*=*NIL* **then** *true*
else if $P(s_0)$="*BLIP* **then** *false* **else** *esteurmă*(*s'*,$P(s_0)$)

Când *s* este finită, generată prin explorarea unei părţi a comportării procesului *P*, recursivitatea se termină.

1.8.3 După

Dacă $s \in urme(P)$ atunci

P/*s* (*P* după *s*)

este un proces ce se comportă identic cu *P* după momentul când *P* s-a angajat în toate acţiunile înregistrate de urma *s*. Dacă *s* nu este o urmă a lui *P*, (*P/s*) nu este definit.

Exemple

X1 (*AVS*/<*mon*>)=(*choc*→*AVS*) □

X2 (*AVS*/<*mon*,*choc*>)=*AVS* □

X3 (*AVC*/<*mon*>^{3})=*STOP* □

X4 Pentru evitarea pierderii unei ciocolate, generată de instalarea lui *AVMMC* (1.1.3 X5, X6), proprietarul aparatului decide să mănânce el ciocolata

(*AVMCCM*/<*choc*>)=*AVS2* □

În reprezentarea arborescentă a lui *P* (fig. 1.1), (*P/s*) semnifică subarborele cu rădăcina în nodul terminal al căii *s*. Astfel, subarborele corespunzător nodului plin ca rădăcină este

(*AVC*/<*in2p*,*mic*,*ex1p*>)

Legile următoare descriu semnificaţia operatorului /. Dacă nu execută nimic, un proces rămâne neschimbat

L1 *P*/<>=*P*

Comportarea lui *P* după angajarea în *s*^*t* este aceeaşi cu a lui (*P/s*) după angajarea în *t*.

L2 *P*/(*s*^*t*)=(*P*/*s*)/*t*

După angajarea într-un eveniment singular *c*, comportarea procesului este definită de această alegere iniţială

L3 (*x*:*B*→*P*(*x*))/<*c*>=*P*(*c*) dacă $c \in B$

Se observă, ca o consecinţă, că /<*c*> este inversul lui *c*→

L3A (*c*→*P*)/<*c*>=*P*

Urmele lui (P/s) le definim astfel

L4 $urme(P/s)=\{t \mid s\hat{}t \in urme(P)\}$ dacă $s \in urme(P)$

Pentru a arăta că un proces nu se opreşte niciodată, este suficient să arătăm că

$P/s \neq STOP$ pentru toate $s \in urme(P)$

O altă proprietate binecunoscută a proceselor este *ciclicitatea*. Un proces este definit ca ciclic dacă din orice situaţie este posibil să se reîntoarcă în starea iniţială

$\forall s{:}urme(P).\ \exists t.\ (P/(s\hat{}t)=P)$

STOP este un proces ciclic banal. Orice alt proces ciclic are proprietatea că nu se opreşte niciodată.

Exemple

X1 Următoarele procese sunt ciclice, (1.1.3 X8, 1.1.2 X2, 1.1.3 X3, 1.1.4 X2)

RUN_A, AVS, $(choc \rightarrow AVS)$, $AVCB$, MM_7 □

X2 Următoarele procese nu sunt ciclice, deoarece nu este posibilă revenirea în starea iniţială (1.1.2 X2, 1.1.3 X3, 1.1.3 X2)

$(mon \rightarrow AVS)$, $(choc \rightarrow AVCB)$, $(pe_loc \rightarrow MM_7)$

De exemplu, starea iniţială a lui $(choc \rightarrow AVCB)$ presupune obţinerea unei singure ciocolate pe când stările ulterioare presupun o alegere între *choc* şi *bonbon*, ceea ce evident este diferit. □

Atenţie. Utilizarea lui / într-un proces definit recursiv are consecinţa nefastă de a invalida gărzile, generându-se pericolul unor soluţii multiple pentru ecuaţii. De exemplu

$X=(a \rightarrow (X/\langle a \rangle))$

este fără gardă şi soluţia poate fi orice proces de forma

$$a \to P$$

oricare ar fi P.

Demonstrație. $(a \to ((a \to P)/\langle a \rangle)) = (a \to P)$ din L3A

Din acest motiv nu vom folosi operatorul / în definirea proceselor recursive.

1.9 Alte operații cu urme

Acest paragraf prezintă alte câteva operații cu urme. Pe moment paragraful poate fi parcurs opțional deoarece vor fi făcute referiri la aceste operații în capitolele următoare.

1.9.1 Schimbare de simbol

Fie f o funcție definită pe o mulțime A de simboluri și cu valori într-o mulțime B. Putem obține din f o funcție f^*, definită pe mulțimea A^* și cu valori în B^*, mulțimi de secvențe, aplicând f fiecărui element din secvența A. De exemplu, *dublu* este o funcție ce își dublează argumentul întreg

$$dublu^*(\langle 1,5,3,1 \rangle) = \langle 2,10,6,2 \rangle$$

O funcție stea este evident distributivă și de aceea strictă

L1 $f^*(\langle\rangle) = \langle\rangle$
L2 $f^*(\langle x \rangle) = \langle f(x) \rangle$
L3 $f^*(s \frown t) = f^*(s) \frown f^*(t)$

Alte legi sunt consecințe evidente

L4 $f^*(s)_0 = f(s_0)$ dacă $s \neq \langle\rangle$
L5 $\#f^*(s) = \#s$

Prezentăm mai jos o lege care pare evidentă, dar nu este adevărată în toate cazurile

$$f^*(s \upharpoonright A) = f^*(s) \upharpoonright f(A)$$

unde $f(A) = \{f(x) \mid x \in A\}$

Un contraexemplu este dat de funcția f unde

$f(b)=f(b)=c$ unde $b\neq c$

Astfel $f^*(\langle b\rangle\upharpoonright\{c\})=f^*(\langle\rangle)$ deoarece $b\neq c$

$=\langle\rangle$ L1

$\neq\langle c\rangle$

$=\langle c\rangle\upharpoonright\{c\}$

$=f^*(\langle b\rangle)\upharpoonright f(\{c\})$ deoarece $f(c)=c$

Totuşi legea este adevărată dacă f este injectivă

L6 $f^*(s\upharpoonright A)=f^*(s)\upharpoonright f(A)$ dacă f este injectivă

1.9.2 Unificare

Fie s o secvenţă ale cărei elemente sunt de asemenea secvenţe. Vom spune că $\frown/s$ este unificarea tuturor elementelor într-o ordine naturală. De exemplu

$\frown/\langle\langle 1,3\rangle,\langle\rangle,\langle 7\rangle\rangle=\langle 1,3\rangle\frown\langle\rangle\frown\langle 7\rangle=\langle 1,3,7\rangle$

Acest operator este distributiv

L1 $\frown/\langle\rangle=\langle\rangle$
L2 $\frown/\langle s\rangle=s$
L3 $\frown/s\frown t=(\frown/s)\frown(\frown/t)$

1.9.3 Întreţeserea

O secventă s se spune că este o întreţesere a secvenţelor t şi u dacă poate fi descompusă într-o serie de subsecvenţe aparţinând lui t şi u.

$s=\langle 1,6,3,1,5,4,2,7\rangle$

este o întreţesere a lui t şi u, unde

$t=\langle 1,6,5,2,7\rangle$ şi $u=\langle 3,1,4\rangle$

O definire recursivă a întreţeserii poate fi dată cu ajutorul următoarelor legi

L1 $\langle\rangle$ *întreţese*$(t,u)\equiv(t=\langle\rangle\wedge u=\langle\rangle)$
L2 s *întreţese*$(t,u)\equiv s$ *întreţese*(u,t)

L3 $(\langle x\rangle\hat{\ }s)$ *întrețese*$(t,u) \equiv (t\neq\langle\rangle \wedge t_0=x \wedge s$ *întrețese*$(t',u))$
$\vee (u\neq\langle\rangle \wedge u_0=x \wedge s$ *întrețese*$(t,u'))$

1.9.4 Indexarea

Dacă $0\leq i<\#s$, folosim notația $s[i]$ pentru al i-lea element al secvenței s, cu definiția dată de L1

L1 $s[0]=s_0 \wedge s[i+1]=s'[i]$ dacă $s\neq\langle\rangle$
L2 $(f^*(s))[i]=f(s[i])$ pentru $i<\#s$

1.9.5 Reversul

Dacă s este o secvență, $\overline{s}$ este secvența ce rezultă prin inversarea ordinii elementelor. De exemplu

$$\overline{\langle 3,5,37\rangle}=\langle 37,5,3\rangle$$

Reversul este complet definit de următoarele legi

L1 $\overline{\langle\rangle}=\langle\rangle$
L2 $\overline{\langle x\rangle}=\langle x\rangle$
L3 $\overline{s\hat{\ }t}=\overline{t}\hat{\ }\overline{s}$

Reversul se bucură de o serie de proprietăți algebrice

L4 $\overline{\overline{s}}=s$

Descoperirea altor proprietăți poate fi făcută de către cititor. Un rezultat util referitor la revers ne spune că $\overline{s}_0$ este ultimul element al unei secvențe, sau mai general

L5 $\overline{s}[i]=s[\#s-i-1]$ pentru $i<\#s$

1.9.6 Selecția

Dacă s este o secvență de perechi, definim $s\downarrow x$ ca fiind rezultatul selectării din s a tuturor perechilor cu prim element x și înlocuirii lor cu al doilea element. Perechea va conține un punct între cele două elemente. Astfel avem

$s=\langle a.7, b.9, a.8, c.0\rangle$

atunci $s\downarrow a=\langle 7,8\rangle$
iar $s\downarrow d=\langle\rangle$

L1 $\langle\rangle\downarrow x=\langle\rangle$
L2 $(\langle y.z\rangle\hat{}t)\downarrow x=t\downarrow x$ dacă $y\neq z$
L3 $(\langle x.z\rangle\hat{}t)\downarrow x=\langle z\rangle\hat{}(t\downarrow x)$

Dacă s nu este o secvență de perechi, $s\downarrow a$ semnifică numărul aparițiilor lui a în s (așa cum rezultă din paragraful 1.6.6).

1.9.7 Compunerea

Notăm cu simbolul ✓ un eveniment ce reprezintă terminarea cu succes a unui proces angajat în el. O primă concluzie este că acest simbol poate apărea doar la sfârșitul unei urme. Fie t o urmă ce reprezintă înregistrarea secvenței evenimentelor care încep după ce s-a terminat s cu succes. Compunerea lui s cu t este notată $(s;t)$. Dacă ✓ nu apare în s, atunci nici t nu poate începe

L1 $s;t=s$ dacă $\neg(\langle✓\rangle$ **in** $s)$

Dacă ✓ este sfârșitul lui s, el este eliminat și t se adaugă rezultatului

L2 $(s\hat{}\langle✓\rangle);t=s\hat{}t$ dacă $\neg(\langle✓\rangle$ **in** $s)$

Simbolul poate fi privit ca un conector de unire al secvențelor s și t. În absența lui din s, secvența t nu poate fi adăugată (L1). Dacă totuși ✓ apare în interiorul unei urme (incorect), stipulăm, pentru compatibilitate, că toate simbolurile după el sunt irelevante și deci trebuie omise

L2A $(s\hat{}\langle✓\rangle\hat{}u);t=s\hat{}t$ dacă $\neg(\langle✓\rangle$ **in** $s)$

Acest fel mai special de compunere se bucură de un număr de proprietăți algebrice. Ca și unificarea, compunerea este asociativă. Spre deosebire de unificare, compunerea este monotonă atât în primul cât și în al doilea argument. De asemenea, este strict monotonă în primul argument și are (✓) ca element unitate la stânga

L3 $s;(t;u)=(s;t);u$
L4A $s\leq t\Rightarrow((u;s)\leq(u;t))$
L4B $s\leq t\Rightarrow((s;u)\leq(t;u))$

L5 $<>;t=<>$
L6 $<\checkmark>;t=t$

Dacă ✓ apare la sfârşitul urmei, $<\checkmark>$ poate fi membrul drept al operatorului compunere

L7 $s;<\checkmark>=s$ dacă $\neg(<\checkmark> \textbf{ in } (\bar{s})')$

1.10 Specificaţii

Specificaţia pentru un produs este descrierea comportării sale. Descrierea este de fapt un predicat ale cărui variabile independente corespund unui aspect observabil al comportării sale. De exemplu, specificarea unui amplificator electronic cu intrarea în gama un volt şi factorul de câştig aproximativ 10 este dată de predicatul

$$AMP10=(0\leq\upsilon\leq\Rightarrow|\upsilon'-10*\upsilon|\leq1)$$

În această specificare, υ semnifică tensiunea de intrare iar υ' tensiunea de ieşire. O semnificaţie clară a variabilelor este esenţială la utilizarea matematicii în alte ştiinţe şi inginerie.

În cazul unui proces, cea mai importantă şi relevantă observaţie a comportării sale este urma evenimentelor până la un moment de timp. Vom folosi variabila *ur* pentru a nota o urmă arbitrară a procesului specificat, aşa cum υ şi υ' semnificau observaţiile privind tensiunile, în exemplul anterior.

Exemple

X1 Patronul unui automat de vânzare nu vrea să aibă vreo pierdere la instalare. De aceea el cere în specificaţie ca numărul ciocolatelor furnizate să nu depăşească pe cel al monedelor introduse în aparat

$$FĂRĂPIERDERI=(\#(ur\upharpoonright\{choc\})\leq\#(ur\upharpoonright\{mon\}))$$ □

În continuare vom folosi abrevierea (introdusă în 1.6.6)

$$ur\downarrow c=\#(ur\upharpoonright\{c\})$$

pentru a nota numărul apariţiilor lui c în ur.

X2 Clientul unui automat de vânzare vrea să se asigure că nu va trebui să introducă noi monezi până când nu primeşts ciocolatele plătite

$ECHIT1=((ur\downarrow mon)\leq(ur\downarrow choc)+1)$ □

X3 **Producătorul unui automat de vânzare trebuie să satisfacă cerinţele atât ale patronilor cât şi ale clienţilor**

$SPECAV=FĂRĂPIERDERI \wedge ECHIT1=(0\leq((ur\downarrow mon)-(ur\downarrow choc)\leq 1)$ □

X4 **O specificare a funcţionării corecte a unui automat interzice introducerea a trei monezi de un penny succesiv**

$CORECTAVC=(\neg(\langle in1p\rangle^3$ **in** $ur)$ □

X5 Specificaţia pentru un automat sigur

$SIGURAVC=(ur\in urme(AVC)\wedge CORECTAVC)$ □

X6 Specificaţia pentru $AVS2$ (1.1.3 X6)

$0\leq((ur\downarrow mon)-(ur\downarrow choc))\leq 2$ □

1.10.1 Satisfacerea

Dacă P este un produs care îndeplineşte specificaţia S spunem că P *satisface* S, notat

P **sat** S

Satisfacerea înseamnă că orice observaţie posibilă a comportării lui P este descrisă de S. Cu alte cuvinte, S este adevărată pentru orice valori date variabilelor, valori generate din observaţiile asupra lui P, sau mai formal

$\forall ur.ur\in urme(P)\Rightarrow S$

De exemplu, următorul tabel prezintă observaţii referitoare la proprietăţile unui amplificator

	1	2	3	4	5
v	0	.5	.5	2	.1
v′	0	5	4	1	3

Toate observațiile cu excepția ultimelor sunt descrise de $AMP10$. A doua și a treia coloană arată faptul că ieșirea amplificatorului nu este complet determinată de intrare. A patra coloană arată că dacă tensiunea de intrare nu este din gama specificată, tensiunea de ieșire poate avea orice valoare fără a contrazice specificațiile. (În acest exemplu simplu s-a ignorat posibilitatea ca o tensiune de intrare prea mare să distrugă amplificatorul.)

Următoarele legi dau proprietățile cele mai generale referitoare la relația *satisface*. Specificația *true*, care presupune inexistența vreunei constrângeri indiferent de observațiile referitoare la produs, este satisfăcută de orice produs. Putem spune că orice produs defect satisface această cea mai slabă specificație

L1 P **sat** *true*

Dacă un produs satisface două specificații diferite, atunci satisface și conjuncția lor

L2A Dacă P **sat** S
și P **sat** T
atunci P **sat** $(S \wedge T)$

Legea L2A se poate generaliza la un număr infinit de conjuncții prin cuantificare. Fie $S(n)$ un predicat cu variabila n

L2 Dacă $\forall n.$ (P **sat** $S(n)$)
atunci P **sat** $(\forall n.\ S(n))$

dacă P nu conține pe n.

Dacă există o implicație logică între specificațiile S și T, atunci orice observație descrisă de S este de asemenea descrisă de T. Totodată, orice produs care satisface S trebuie să satisfacă și specificația mai slabă T

L3 Dacă P **sat** S
și $S \Rightarrow T$
atunci P **sat** T

Prin prisma ultimei legi vom scrie în cazul unor reguli, în care intervin implicații de forma $S \Rightarrow T$

P **sat** S
$\Rightarrow T$

ca o prescurtare pentru

P **sat** S
$S \Rightarrow T$
P **sat** T din L3

Legile şi explicaţiile date mai sus pot fi aplicate oricărui produs şi oricărei specificaţii. În următorul paragraf se vor prezenta legile specifice proceselor.

1.10.2 Reguli

În proiectarea unui produs trebuie să ne asigurăm că acesta va satisface specificaţiile. Responsabilitatea poate fi asumată folosind raţionamente din diverse domenii ale matematicii, ca de exemplu geometria diferenţială, calcul diferenţial şi integral. În prezentul paragraf se prezintă un set de legi ce folosesc raţionamente algebrice, logice, permiţând demonstrarea îndeplinirii specificaţiei S de către procesul P.

Vom scrie $S(ur)$ pentru a arăta că specificaţia are pe *ur* drept variabilă liberă. Un alt motiv pentru care *ur* este explicită este posibilitatea de a o substitui cu expresii complicate, ca de exemplu *ur*". Este evident că atât S cât şi $S(ur)$ au şi alte variabile în afară de *ur*

Orice observaţie relativă la procesul *STOP* va fi întotdeauna urma vidă, deoarece procesul nu face nimic

L4A *STOP* **sat** $(ur=\langle\rangle)$

Urma procesului $(c \rightarrow P)$ este iniţial vidă. Orice urmă ulterioară începe cu c iar coada este o urmă a lui P. De asemenea coada trebuie să poată fi descrisă de orice specificaţie a lui P

L4B Dacă P **sat** $S(ur)$
atunci $(c \rightarrow P)$ **sat** $(ur=\langle\rangle \vee (ur_0=c \wedge S(ur')))$

Un corolar al acestei legi tratează prefixul dublu

L4C Dacă P **sat** $S(ur)$
atunci $(c \rightarrow d \rightarrow P)$ **sat** $(ur \leq \langle c,d\rangle \vee (ur \geq \langle c,d\rangle \wedge S(ur'')))$

Alegerea binară este similară prefixului, dar urma poate începe cu oricare din cele două evenimente alternative, iar coada este descrisă de specificaţia alternativei alese

Alegerea binară este similară prefixului, dar urma poate începe cu oricare din cele două evenimente alternative, iar coada este descrisă de specificația alternativei alese

L4D Dacă P **sat** $S(ur)$
și Q **sat** $T(ur)$
atunci $(c \rightarrow P | d \rightarrow Q)$ **sat** $(ur=\langle\rangle \vee (ur_0=c \wedge S(ur')) \vee (ur_0=d \wedge T(ur')))$

Toate legile descrise mai sus sunt cazuri particulare ale legii alegerii generale

L4 Dacă $\forall x \in B.\ (P$ **sat** $S(ur,x))$
atunci $(x{:}B \rightarrow P(x))$ **sat** $(ur=\langle\rangle \vee (ur_0 \in B \wedge S(ur', ur_0)))$

Legea pentru operatorul *după* este surprinzător de simplă. Dacă ur este o urmă a lui (P/s), $s \frown ur$ este o urmă a lui P și de aceea trebuie să poată fi descrisă de orice specificație pentru P

L5 Dacă P **sat** $S(ur)$
și $s \in urme(P)$
atunci (P/s) **sat** $S(s \frown ur)$

În fine, avem nevoie de o lege care să stabilească corectitudinea procesului definit recursiv

L6 Dacă $F(X)$ este cu gardă
și $STOP$ **sat** S
și $((X$ **sat** $S) \Rightarrow (F(X)$ **sat** $S))$
atunci $(\mu X.F(X))$ **sat** S

Prin inducție putem deduce că

$F^n(STOP)$ **sat** S

Deoarece F este cu gardă, $F^n(STOP)$ descrie complet cel puțin primii n pași din comportarea lui $\mu X.F(X)$. Astfel, orice urmă a lui $\mu X.F(X)$ este o urmă pentru $F^n(STOP)$, oricare ar fi n. Această urmă trebuie, de asemenea, să satisfacă aceeași specificație S ca și $F^n(STOP)$. O justificare mai riguroasă va fi dată în paragraful 2.8 cu ocazia demonstrației matematice de la sfârșit.

Exemplu

X1 Vrem să dovedim că (1.1.2 X2, 1.10 X3)

AVS **sat** $SPECAV$

Demonstrație. (1) $STOP$ **sat** $(ur=\langle\rangle)$ L4A

$\Rightarrow 0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1$

deoarece $(\langle\rangle \downarrow mon) = (\langle\rangle \downarrow choc) = 0$

Concluzia rezultă folosind L3

(2) Presupunem X **sat** $(0 \leq ((ur \downarrow mon) - (ur \downarrow choc)) \leq 1)$

Deci $(mon \rightarrow choc \rightarrow X)$ **sat** $(ur \leq \langle mon, choc\rangle$

$\vee (ur \geq \langle mon, choc\rangle$

$\wedge 0 \leq ((ur'' \downarrow mon) - (ur'' \downarrow choc) \leq 1))$ L4C

$\Rightarrow 0 \leq ((ur \downarrow mon) - (ur \downarrow choc)) \leq 1$

deoarece $\langle\rangle \downarrow mon = \langle\rangle \downarrow choc = \langle mon\rangle \downarrow choc = 0$

și $\langle mon\rangle \downarrow mon = (\langle mon, choc\rangle \downarrow mon) = \langle mon, choc\rangle \downarrow choc = 1$

și $ur \geq \langle mon, choc\rangle \Rightarrow (ur \downarrow mon = ur'' \downarrow mon + 1 \wedge ur \downarrow choc = ur'' \downarrow choc + 1)$

Concluzia rezultă din L3 și L6 □

Faptul că un proces satisface specificațiile nu înseamnă automat că este util necesităților cerute. De exemplu, avem

$ur = \langle\rangle \Rightarrow 0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1$

și putem deduce din L3 și L4A că

$STOP$ **sat** $0 \leq (ur \downarrow mon) - (ur \downarrow choc) \leq 1$

Cu toate acestea *STOP* nu poate fi folosit ca un automat de vânzare, nici pentru patron, nici pentru client. Este clar că evită să facă ceva greșit, dar asta nu înseamnă nimic. Putem spune că *STOP* satisface orice specificație satisfăcătoare oricărui proces.

Din fericire, este evident că *AVS* nu se poate opri vreodată. De fapt, orice proces definit în termenii prefixului, alegerii și recursivității cu gardă nu se va opri niciodată. Singura modalitate de a descrie un proces cu posibilitatea de a se opri este de a include explicit în descrierea sa *STOP* sau $(x{:}B \rightarrow P(x))$, unde B este mulțime vidă. Prin evitarea acestor cazuri putem scrie procese care garantat nu se vor opri. Totuși, după introducerea concurenței, în următorul

capitol, vom vedea că aceste singure premise nu sunt suficiente. Modalitatea generală de a demonstra că un proces nu se opreşte este dată în paragraful 3.7.

2 Concurență

2.1 Introducere

Un proces este definit prin descrierea întregii sale comportări posibile. De multe ori, există o alegere între diferitele acțiuni, de exemplu introducerea unei monezi de doi penny sau a uneia de un penny într-un AVC (1.1.3 X4). În fiecare caz, alegerea primului eveniment poate fi controlată de mediul în care evoluează procesul. De exemplu, cumpărătorul poate selecta ce monedă introduce în AVC. Din fericire, mediul unui proces poate fi descris de asemenea ca un proces cu comportarea definită prin notațiile introduse. Aceasta permite investigarea comportării unui sistem complet format din proces și mediul său, cei doi interacționând concurent. Sistemul poate fi privit ca un proces a cărui comportare este definită în termenii comportărilor asigurate pentru cele două procese componente. Sistemul poate fi plasat la rândul lui într-un mediu mai cuprinzător ş.a.m.d. De fapt, este bine de înțeles că putem şterge diferența între procese, medii şi sisteme. Toate sunt procese a căror comportare poate fi imaginată, descrisă, înregistrată şi analizată într-o manieră simplă şi omogenă.

2.2 Interacțiune

Când două procese sunt puse să evolueze concurent, intenția este ca ele să interacționeze. Aceste interacțiuni pot fi privite ca evenimente ce necesită participarea simultană a celor două procese. Pentru cele ce urmează, să ne îndreptăm atenția spre astfel de evenimente şi să le ignorăm pe celelalte. Vom presupune totodată că alfabetele celor două procese sunt identice. De asemenea, fiecare eveniment care apare trebuie să fie un eveniment posibil în comportarea independentă a fiecărui proces separat. De exemplu, o ciocolată poate fi extrasă dintr-un AV numai când cumpărătorul doreşte şi AV este pregătit să o facă. Fie P şi Q două procese cu acelaşi alfabet, atunci intoducem

$P \| Q$

pentru a desemna procesul care se comportă ca sistemul compus din procesele P şi Q ce interacţionează sincronizându-se în anumite momente, conform celor expuse mai sus.

Exemple

X1 Un client pofticios este fericit de a obţine o bomboană sau o ciocolată fără a plăti. Totuşi, el va plăti, fără tragere de inimă, o monedă, după care insistă să primească ciocolată

$$CLPOF = (bonbon \to CLPOF \mid choc \to CLPOF \mid mon \to choc \to CLPOF)$$

Când acest consumator se găseşte faţă în faţă cu *AVCB* (1.1.3 X3) este frustrat, deoarece automatul nu permite extragerea produselor înaintea plăţii. Pe de altă parte, *AVCB* nu va da nici o bomboană, pentru că consumatorul a plătit o ciocolată

$$(CLPOF \| AVCB) = \mu X.(mon \to choc \to X)$$

Acest exemplu arată cum un proces definit ca o compoziţie a două subprocese poate fi rescris ca un proces fără operatorul $\|$. □

X2 Un consumator zăpăcit doreşte o napolitană mare astfel că pune o monedă în *AVC*. Nu observă dacă a introdus un penny sau doi penny

$$ZĂPĂCIT = (in2p \to mare \to ZĂPĂCIT \mid in1p \to mare \to ZĂPĂCIT)$$

Din nefericire, *AVC* nu poate elibera o napolitană mare pentru un penny

$$(ZĂPĂCIT \| AVC) = \mu X.(in2p \to mare \to X \mid in1p \to STOP)$$

STOP-ul care intervine după primul eveniment $in1p$ se mai numeşte *blocaj*. De fapt, este situatia când fiecare proces component este pregătit să se angajeze în acţiuni ulterioare diferite. Deoarece nu se pot înţelege care acţiune să fie aceasta se ajunge la blocaj. □

Exemplele anterioare pun în evidenţă abdicarea de la abstractizare şi obiectivitate. Este important de rememorat că evenimentele sunt nişte tranziţii neutre care pot fi observate şi înregistrate de oricine, fără a cunoaşte, de exemplu, a priori, cum este o napolitană mare, mică, o ciocolată sau bomboană. Special s-a ales alfabetul format din evenimente relevante pentru a exclude

stări emoționale sau alte aspecte. Dacă dorim putem introduce evenimente pentru modelarea schimbării stărilor interne, ca în 2.3 X1.

2.2.1 Legi

Legile guvernând comportarea lui $(P||Q)$ sunt simple și regulate. Prima exprimă simetria între proces și mediu

L1 $P||Q = Q||P$

Următoarea lege arată că nu contează ordinea în care interacționează procesele

L2 $P||(Q||R) = (P||Q)||R$

Interacțiunea cu procesul aflat în blocaj duce la blocaj. O interacțiune cu procesul $RUN_{\alpha P}$ (1.1.3 X8) nu presupune vreo restricție ulterioară pentru P

L3A $P||STOP_{\alpha P}= STOP_{\alpha P}$

L3B $P||RUN_{\alpha P} =P$

Următoarele legi arată cum o pereche de procese se angajează simultan în aceeași acțiune sau se blochează dacă prima acțiune nu este comună

L4A $(c\rightarrow P)||(c\rightarrow Q)=(c\rightarrow(P||Q))$

L4B $(c\rightarrow P)||(d\rightarrow Q)=STOP$ dacă $c\neq d$

Aceste legi se generalizează la cazurile când unul sau ambele procese oferă o alegere pentru evenimentul inițial. Numai evenimentele pe care le oferă amândouă procesele (comune) sunt posibile de a fi angajate când procesele se combină interacționând

L4 $(x:A\rightarrow P(x))||(y:B\rightarrow Q(y))=(z:(A\cap B)\rightarrow(P(z)||Q(z)))$

Această lege permite unui sistem definit în termeni de concurență să fie transformat într-o descriere alternativă fără concurență

Exemplu

X1 Fie $P=(a\rightarrow b\rightarrow P|b\rightarrow P)$

și $Q=(a\rightarrow(b\rightarrow Q|c\rightarrow Q))$

Atunci $(P||Q)=a\rightarrow((b\rightarrow P)||(b\rightarrow Q|c\rightarrow Q))$ din L4A

$=a\rightarrow(b\rightarrow(P||Q))$ din L4A

$=\mu X.(a \to b \to X)$ deoarece recursivitatea este cu gardă

□

2.2.2 Implementare

Implementarea operatorului || se bazează pe L4

$$intersect(P,Q)=\lambda z.\textbf{if } P(z)=\text{"}BLIP \vee Q(z)=\text{"}BLIP \textbf{ then } \text{"}BLIP$$
$$\textbf{else } intersect(P(z),Q(z))$$

2.2.3 Urme

Deoarece fiecare acțiune a lui $(P||Q)$ necesită participarea simultană atât a lui P cât și a lui Q, fiecare secvență de astfel de acțiuni trebuie să fie posibilă pentru ambii operanzi. Pentru același motiv $/s$ este distributivă față de ||

L1 $urme(P||Q)=urme(P)\cap urme(Q)$

L2 $(P||Q)/s=(P/s)||(Q/s)$

2.3 Concurență

Operatorul descris în paragraful anterior poate fi generalizat la cazul când operanzii P și Q au alfabete diferite

$$\alpha P \neq \alpha Q$$

Când procesele evoluează concurent, evenimentele care sunt în ambele alfabete (așa cum s-a arătat mai sus) necesită participarea simultană a lui P și Q. Evenimentele care aparțin exclusiv lui P nu-l interesează pe Q, care nu are cum să le observe. Astfel de evenimente pot apărea independent de Q când P se angajează în ele. Similar, Q se poate angaja în evenimente exclusiv ale lui. De aceea, mulțimea evenimentelor care sunt logic posibile pentru întregul sistem este desigur reuniunea alfabetelor proceselor componente

$$\alpha(P||Q)=\alpha P \cup \alpha Q$$

Acest operator de compunere este un exemplu mai rar de operator cu operanzi cu alfabete diferite iar rezultatul are un al 3-lea alfabet. Totuși, când cei doi operanzi au același alfabet, rezultatul are și el același alfabet și $(P||Q)$ are evident semnificația anterioară

Exemple

X1 Fie $\alpha AVSUNET=\{mon,choc,bing,bang,bonbon\}$
unde *bing* este sunetul unei monezi căzând în cutia de bani a automatului de vânzare pentru care se iau în considerare și sunete
și *bang* este sunetul făcut de automat la terminarea unei operații

Automatul nu eliberează bomboane (s-au terminat)

$$AVSUNET=(mon\rightarrow bing\rightarrow choc\rightarrow bang\rightarrow AVSUNET)$$

Dacă consumatorul preferă bomboane n-are decât să blesteme, dar trebuie să ia numai ciocolate

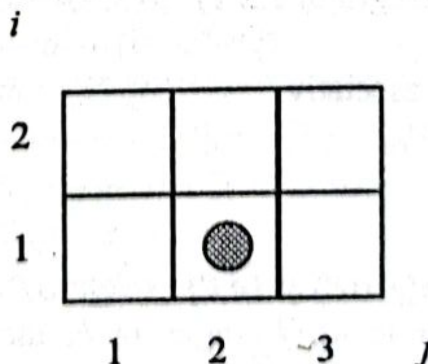
$$\alpha CONS=\{mon,choc,blestem,bonbon\}$$
$$CONS=(mon\rightarrow(bonbon\rightarrow CONS|blestem\rightarrow choc\rightarrow CONS))$$

Rezultatul activității concurente a celor două procese este

$$(AVSUNET||CONS)=\mu X.(mon\rightarrow(bing\rightarrow blestem\rightarrow choc\rightarrow bang\rightarrow X$$
$$|blestem\rightarrow bing\rightarrow choc\rightarrow bang\rightarrow X))$$

Trebuie observat că *bing* poate fi intervertit cu *blestem*. Aceste evenimente pot apare în succesiune sau simultan, caz în care nu contează ordinea în care se înregistrează. De notat că formula matematică nu reprezintă în nici un fel faptul că clientul preferă o bomboană în loc de a exprima un blestem. Formula este o abstractizare a realității care ignoră emoțiile umane și se concentrează numai pe posibilitățile de apariție sau nu a evenimentelor din alfabete și nu a evenimentelor dorite sau nu, posibile sau nu. □

X2



Un marker porneşte din poziţia arătată pe o suprafaţă riglată şi se poate mişca *sus*, *jos*, *dreapta*, *stânga*

Fie $\alpha P=\{sus, jos\}$
$P=(sus\rightarrow jos\rightarrow P)$
$\alpha Q=\{stânga, dreapta\}$
$Q=(dreapta\rightarrow stânga\rightarrow Q|stânga\rightarrow dreapta\rightarrow Q)$

Comportarea markerului poate fi definită de

$P||Q$

În acest exemplu, alfabetele αP şi αQ *nu* au vreun element în comun. De asemenea, mişcările markerului sunt determinate de întreţeserea acţiunilor din procesul P cu cele din Q. Astfel de întreţeseri ale acţiunilor sunt foarte greu de descris fără concurenţă. De exemplu, fie R_{ij} procesul care semnifică comportarea markerului (X2) cînd se găseşte în linia i şi coloana j de pe suprafaţă, $i\in\{1, 2\}, j\in\{1, 2, 3\}$.

Atunci $(P||Q) = R_{12}$, unde
$R_2 =(jos\rightarrow R_{11}|dreapta\rightarrow R_{22})$
$R_{11}=(sus\rightarrow R_{21}|dreapta\rightarrow R_{12})$
$R_{22}=(jos\rightarrow R_{12}|stânga\rightarrow R_{21}|dreapta\rightarrow R_{23})$
$R_{12}=(sus\rightarrow R_{22}|stânga\rightarrow R_{11}|dreapta\rightarrow R_{13})$
$R_{23}=(jos\rightarrow R_{13}|stânga\rightarrow R_{22})$
$R_{13}=(sus\rightarrow R_{23}|stânga\rightarrow R_{12})$ □

2.3.1 Legi

Primele trei legi pentru forma extinsă de concurenţă sunt aceleaşi cu legile pentru interacţiune (paragraful 2.2.1)

L1,2 || este simetric şi asociativ.
L3A $P||STOP_{\alpha P}=STOP_{\alpha P}$
L3B $P||RUN_{\alpha P}=P$

Fie $a\in(\alpha P-\alpha Q)$, $b\in(\alpha Q-\alpha P)$ şi $\{c,d\}\subseteq(\alpha Q\cap\alpha P)$. Următoarele legi arată cum P se angajează singur în a, Q singur în b, iar c şi d necesită participarea simultană a lui P şi Q

L4A $(c\rightarrow P)||(c\rightarrow Q)=c\rightarrow(P||Q)$

L4B $(c \to P) \| (d \to Q) = STOP$ dacă $c \neq d$

L5A $(a \to P) \| (c \to Q) = a \to (P \| (c \to Q))$

L5B $(c \to P) \| (b \to Q) = b \to ((c \to P) \| Q))$

L6 $(a \to P) \| (b \to Q) = a \to (P \| (b \to Q)) | b \to ((a \to P) \| Q))$

Aceste legi pot fi generalizate pentru operatorul alegere generală

L7 Fie $P = (x{:}A \to P(x))$

și $Q = (y{:}B \to Q(y))$

Atunci $(P \| Q) = (z{:}C \to P' \| Q')$

unde $C = (A \cap B) \cup (A - \alpha Q) \cup (B - \alpha P)$

și $P' = P(z)$ dacă $z \in A$

$= P$ altfel

$Q' = Q(z)$ dacă $z \in B$

$= Q$ altfel

Legile de mai sus permit unui proces definit prin concurență să fie redefinit fără a folosi acest operator, ca în următoarele exemple

Exemple

X1 Fie $\alpha P = \{a, c\}$, $\alpha Q = \{b, c\}$

Fie $P = (a \to c \to P)$

Fie $Q = (c \to b \to Q)$

De aceea $P \| Q = (a \to c \to P)(c \to b \to Q)$ din definiție

$= a \to ((c \to P) \| (c \to b \to Q))$ din L5

$= a \to c \to (P \| (b \to Q))$ din L4...(1)

De asemenea $P \| (b \to Q) = (a \to (c \to P) \| (b \to Q)$

$| b \to (P \| Q))$ din L6

$= (a \to b \to ((c \to P) \| Q)$

$| b \to (P \| Q))$ din L5

$= (a \to b \to c \to (P \| (b \to Q))$ din L4

$| b \to a \to c \to (P \| (b \to Q)))$ (1) de mai sus

$= \mu X.(a \to b \to c \to X$ deoarece este

$| b \to a \to c \to X)$ cu gardă

De aceea $(P \| Q) = a \to c \to \mu X.(a \to b \to c \to X$

$| b \to a \to c \to X))$ din (1) de mai sus

□

2.3.2 Implementare

Implementarea operatorului || este derivată direct din legea L7. Alfabetele operanzilor sunt reprezentate de liste finite de simboluri, *A* și *B*. Testul de apartenență folosește funcția *estemembru(x,A)* definită în paragraful 1.7.

(*P*||*Q*) este implementat prin apelul funcției
concurent(*P*,α*P*,α*Q*,*Q*)

definită astfel

concurent(P,A,B,Q)=aux(P,Q)

unde *aux(P,Q)*=
λ*x*.if *P*="*BLIP* **sau** *Q*="*BLIP* **then** "*BLIP*
else if *estemembru(x,A)* ∧ *estemembru(x,B)* **then** *aux*(*P*(*x*), *Q*(*x*))
else if *estemembru(x, A)* **then** *aux*(*P*(*x*), *Q*)
else if *estemembru(x, B)* **then** *aux*(*P*, *Q*(*x*))
else "*BLIP*

2.3.3 Urme

Fie *t* urma lui (*P*||*Q*). Orice eveniment din *t* care aparține alfabetului lui *P* este un eveniment în evoluția lui *P*. Orice eveniment din *t* care nu aparține lui α*P* a apărut fără participarea lui *P*. Astfel ($t \upharpoonright \alpha P$) este o urmă formată din evenimente în care a participat efectiv *P* și de aceea o urmă a lui *P*. Analog, ($t \upharpoonright \alpha Q$) este o urmă a lui *Q*. Mai mult, fiecare eveniment din *t* trebuie să fie ori în α*P* ori în α*Q*. Acest raționament sugerează legea

L1 $urme(P||Q)=\{t \mid (t \upharpoonright \alpha P) \in urme(P) \wedge (t \upharpoonright \alpha Q) \in urme(Q) \wedge t \in (\alpha P \cup \alpha Q)^*\}$

Următoarea lege arată cum operatorul / *s* se distribuie în compunerea paralelă

L2 $(P||Q)/s=(P/(s \upharpoonright \alpha P))||(Q/(s \upharpoonright \alpha Q))$

Când $\alpha P = \alpha Q$ avem

$$s \upharpoonright \alpha P = s \upharpoonright \alpha Q = s$$

și aceste legi sunt identice cu cele din paragraful 2.2.3.

Exemple

X1 Vezi 2.3 X1.

Fie $t1=\langle mon,bing,blestem\rangle$

atunci $t1\upharpoonright\alpha AVSUNET=\langle mon,bing\rangle$ este în *urme*(*AVSUNET*)

şi $t1\upharpoonright\alpha CONS=\langle mon,blestem\rangle$ este în *urme*(*CONS*)

De aceea $t1\in urme(AVSUNET\|CONS)$

Un raţionament similar ne arată că

$$\langle mon,blestem,bing\rangle\in urme(AVSUNET\|CONS)$$

Cele de mai sus confirmă că evenimentele *blestem* şi *bing* pot fi intervertite în orice ordine. Ele pot să apară chiar simultan dar atunci nu le putem înregistra decât tot într-o succesiune. □

În concluzie, urma lui $(P\|Q)$ este o întreţesere a urmei lui P cu urma lui Q, evenimentele comune aparînd numai o dată. Dacă $\alpha P\cap\alpha Q=\{\}$, urmele sunt întreţeseri efective (paragraful 1.9.3.), conform cu paragraful 2.3 X2. La cealaltă extremă, când $\alpha P=\alpha Q$ fiecare eveniment aparţine ambelor alfabete şi sensul lui $(P\|Q)$ este exact cel definit pentru interacţiune (paragraful 2.2).

L3A Dacă $\alpha P\cap\alpha Q=\{\}$

$urme(P\|Q)=\{s \mid \exists\, t{:}urme(P).\exists\, u{:}urme(Q).s \text{ întreţese } (t,u)\}$

L3B Dacă $\alpha P=\alpha Q$

$urme(P\|Q)=urme(P)\cap urme(Q)$

2.4 Reprezentări grafice

Procesul P cu alfabetul $\{a,b,c\}$ este figurat ca un dreptunghi etichetat P bordat de un număr de linii, fiecare etichetată cu un eveniment diferit din alfabetul său (fig. 2.1). Analog Q, cu alfabetul $\{b,c,d\}$ este reprezentat ca în fig. 2.2.

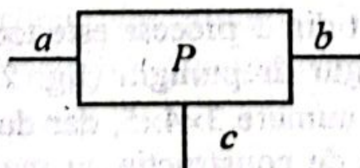


Figura 2.1

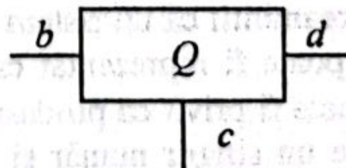


Figura 2.2

Când aceste două procese sunt puse să evolueze concurent, sistemul rezultat poate fi desenat ca o rețea în care liniile etichetate la fel sunt conectate iar liniile etichetate cu evenimente proprii proceselor rămân libere (fig. 2.3.). Un al 3-lea proces *R*, unde $\alpha R=\{c,e\}$, poate fi adăugat ca în fig. 2.4.

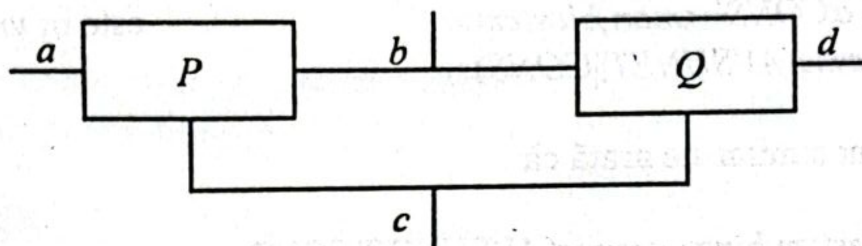


Figura 2.3

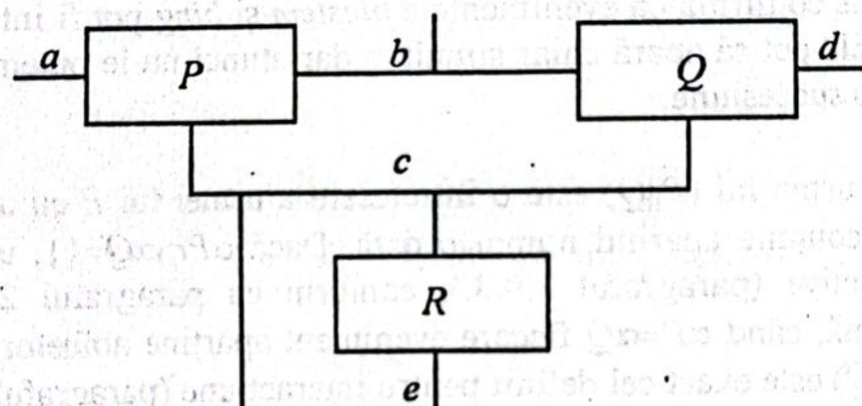


Figura 2.4

Se observă că evenimentul *c* necesită participarea tuturor celor trei procese, *b* necesită participarea lui *P* și *Q*, în timp ce evenimentele proprii *a*,*d*,*e* privesc un singur proces. Reprezentările de acest fel vor fi numite diagrame de conexiune.

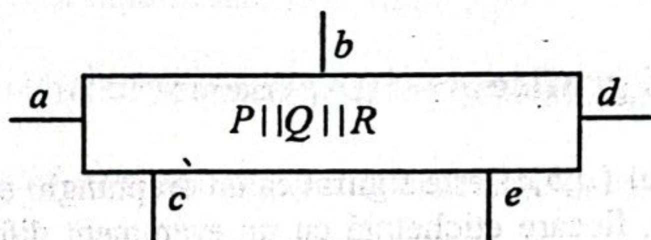


Figura 2.5

Trebuie reamintit că un sistem construit din 3 procese este totuși un singur proces și poate fi reprezentat ca un singur dreptunghi (fig. 2.5). Astfel, numărul 60 poate fi privit ca produsul a trei numere 3×4×5, dar după ce l-am obținut rămâne un singur număr și maniera de construcție nu mai este relevantă sau observabilă.

2.5 Exemplu: Problema mesei celor cinci filozofi

În antichitate, un filantrop bogat a donat unei Academii cele necesare pentru a găzdui 5 filozofi eminenţi. Fiecare filozof avea o cameră în care îşi putea desfăşura activitatea sa de gândire. Exista de asemenea o sală de mese comună, cu o masă rotundă şi cinci scaune etichetate cu numele filozofului ce urma să se aşeze pe el. Numele filozofilor erau FIL_0,FIL_1,FIL_2,FIL_3,FIL_4, şi ei erau dispuşi în sens trigonometric în jurul mesei. În stânga fiecărui filozof era aşezată o furculiţă iar în centrul mesei era un castron mare cu spaghetti, ce era continuu umplut.

Un filozof îşi petrecea cea mai mare parte a timpului gândind. Când i se făcea foame, se ducea în sala de mese, se aşeza pe propriul său scaun, îşi lua propria furculiţă din stânga şi o înfigea în castronul cu spaghetti. Dar natura încâlcită a spaghettilor face să fie necesară încă o furculiţă. De aceea filozoful avea nevoie şi de furculiţa din dreapta. Când filozofii ce mâncau terminau, puneau jos ambele furculiţe, se ridicau şi se duceau în camerele lor pentru a reflecta din nou. Desigur, o furculiţă putea fi folosită numai de un filozof la un moment dat. Dacă alt filozof ar fi dorit-o, el trebuia să aştepte până când devenea disponibilă.

2.5.1 Alfabete

Vom construi acum un model matematic al acestui sistem. Pentru început trebuie să selectăm mulţimea de evenimente relevante. Pentru FIL_i, mulţimea este definită astfel

$$\alpha FIL_i=\{i.aşează, i.ridică,$$
$$i.preia_furc.i, i.preia_furc.(i\oplus 1),$$
$$i.elib_furc.i, i.elib_furc.(i\oplus 1)\}$$

unde $\oplus$ este suma modulo 5, astfel că $i\oplus 1$ identifică vecinul din dreapta al filozofului i.

Se observă că alfabetele filozofilor sunt disjuncte. Nu există nici un eveniment în care să participe împreună, astfel că nu există nici o modalitate de a interacţiona sau comunica între ei - o comportare reflectată realist pentru atitudinea filozofilor din acele vremuri.

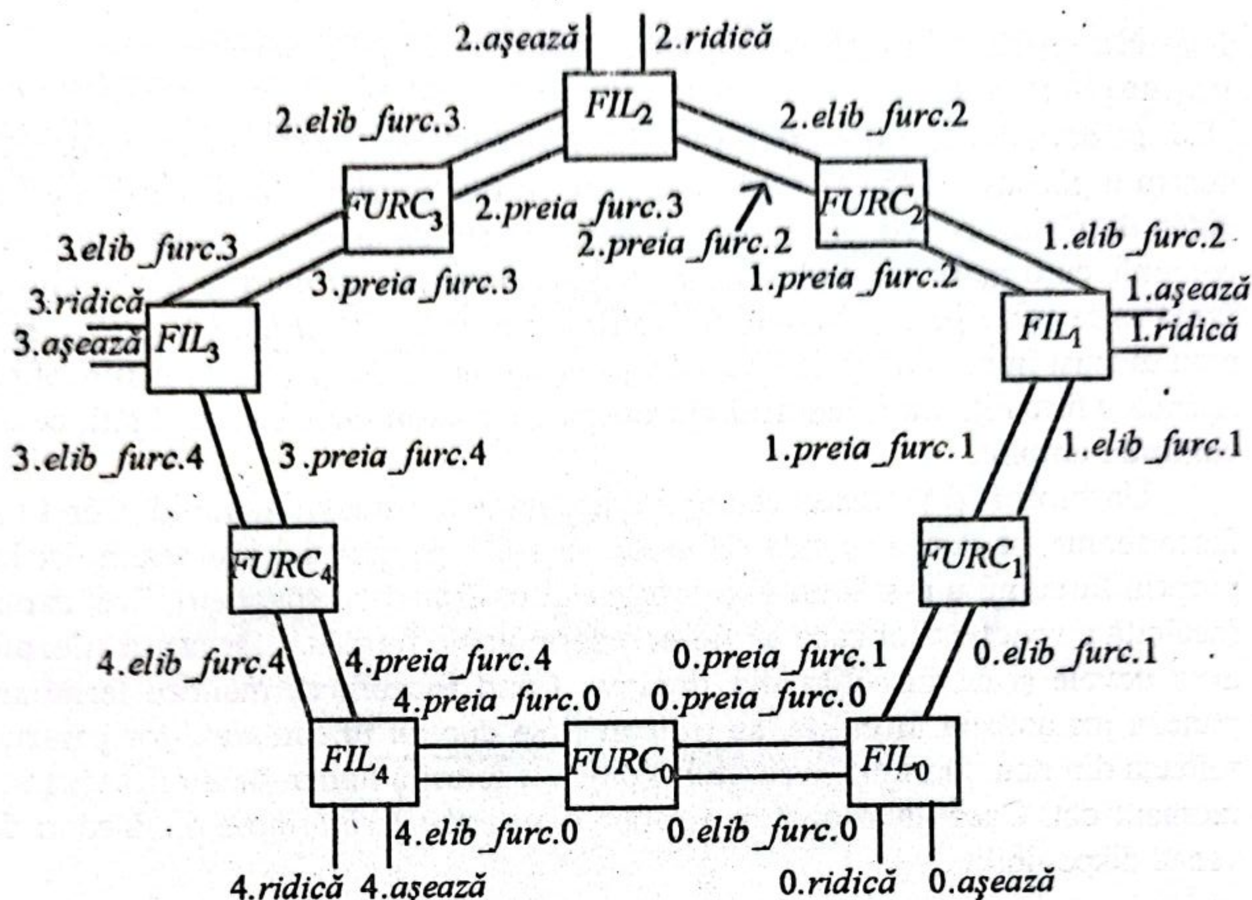


Figura 2.6

Ceilalți actori în mica noastră dramă sunt cele cinci furculițe, fiecare purtând același număr ca și filozoful în stânga căruia se află. O furculiță este preluată sau pusă înapoi pe masă de acest filozof sau de vecinul lui. Alfabetul furculiței *i* este definit astfel

$$\alpha FURC_i = \{i.preia_furc.i, (i\ominus 1).preia_furc.i,$$
$$i.elib_furc.i, (i\ominus 1).elib_furc.i\}$$

unde ⊖ înseamnă scădere modulo 5.

Astfel, fiecare eveniment exceptând ridicatul sau așezatul pe scaun, necesită participarea exact a doi actori vecini, un filozof și o furculiță, așa cum se arată în fig. 2.6.

2.5.2 Comportarea

În afară de starea de a reflecta și a mânca, pe care le ignorăm, existența fiecărui filozof este descrisă de repetarea unui ciclu de șase evenimente

$$FIL_i = (i.așează \rightarrow i.preia_furc.i \rightarrow i.preia_furc.(i\oplus 1) \rightarrow$$
$$i.elib_furc.i \rightarrow i.elib_furc.(i\oplus 1) \rightarrow i.ridică \rightarrow FIL_i)$$

Rolul unei furculițe este simplu, ea fiind mereu ridicată și pusă jos de unul din filozofii alăturați (aceeași în ambele situații)

$$FURC_i=(i.preia_furc.i \to i.elib_furc.i \to FURC_i$$
$$|(i\ominus 1).preia_furc.i \to (i\ominus 1).elib_furc.i \to FURC_i\ )$$

Descrierea activităților din cadrul Academiei este compunerea concurentă a comportărilor elementelor sale

$$FIL_5=(FIL_0\|FIL_1\|FIL_2\|FIL_3\|FIL_4)$$
$$FURC_5=(FURC_0\|FURC_1\|FURC_2\|FURC_3\|FURC_4)$$
$$ACADEMIE=(FIL_5\|FURC_5)$$

O variantă interesantă a acestei istorii permite filozofilor să-și ridice furculițele în orice ordine vor sau să le așeze în orice ordine. Să considerăm comportarea mâinilor fiecărui filozof. Fiecare mână este capabilă să ridice furculița corespunzătoare, dar ambele mâini (ale aceluiași filozof) sunt necesare în situațiile de așezare sau ridicare de pe scaun

$$\alpha STÂNGA_i=\{i.așează, i.ridică, i.preia_furc.i, i.elib_furc.i\}$$
$$\alpha DREAPTA_i=\{i.așează, i.ridică, i.preia_furc.(i\oplus 1), i.elib_furc.(i\oplus 1\}$$
$$STÂNGA_i=(i.așează \to i.preia_furc.i \to i.elib_furc.i \to i.ridică \to STÂNGA_i)$$
$$DREAPTA_i=(i.așează \to i.preia_furc.(i\oplus 1) \to i.elib_furc.(i\oplus 1) \to i.ridică \to DREAPTA_i)$$

$$FIL_i=STÂNGA_i\|DREAPTA_i$$

Sincronizarea pe evenimentele *așează* și *ridică* de pe scaun între $STÂNGA_i$ și $DREAPTA_i$ ne asigură că nici o furculiță nu va fi ridicată decât când filozoful corespunzător este așezat. Astfel operațiile cu cele două furculițe adiacente unui filozof sunt întrețesute arbitrar.

În altă variantă a acestei istorii fiecare furculiță poate fi ridicată și așezată de mai multe ori în timp ce un filozof stă la masă. De aceea comportarea mâinilor este modificată pentru a conține o buclă, de exemplu

$$STÂNGA_i=(i.așează \to \mu X.(i.preia_furc.i \to i.elib_furc.i \to X$$
$$|i.ridică \to STÂNGA_i\ ))$$

2.5.3 Blocaj

Când s-a construit modelul matematic, a apărut următoarea primejdie. Să presupunem că toți filozofii devin flămânzi cam în același timp. Ei se vor așeza și ridica propria furculiță (stânga). Toți vor căuta să ridice și cealaltă furculiță

care nu e disponibilă. În această situație toți vor flămânzi și mai tare, inevitabil. Cu toate că orice proces este capabil de o acțiune ulterioară, evident nu există nici una asupra căreia două procese să se înțeleagă să o realizeze.

Totuși, istoria nu se termină așa trist. Odată pericolul detectat s-au găsit căi pentru evitarea lui. De exemplu, unul dintre filozofi ar putea ridica întâi furculița din dreapta lui sau s-ar putea achiziționa o furculiță (cinci furculițe în plus ar fi prea scump).

Soluția finală adoptată a fost numirea unui valet a cărui sarcină era să supravegheze (să conducă) așezarea și ridicarea fiecărui filozof de pe scaunul său. Alfabetul său este

$$\bigcup_{i=0}^{4} \{i.\textit{așează}, i.\textit{ridică}\}$$

Acestui valet i s-au dat instrucțiuni secrete de a nu permite să se așeze mai mult de patru filozofi odată la masă. Comportarea lui este cel mai simplu definită recursiv mutual.

Fie $R=\bigcup_{i=0}^{4} \{i.\textit{ridică}\}$, $A=\bigcup_{i=0}^{4} \{i.\textit{așează}\}$

$VALET_i$ definește comportarea valetului cu i filozofi așezați

$$VALET_0=(x{:}A\rightarrow VALET_1)$$
$$VALET_j=(x{:}A\rightarrow VALET_{j+1}|y{:}R\rightarrow VALET_{j-1}) \qquad j\in\{1,2,3\}$$
$$VALET_4=(y{:}R\rightarrow VALET_3)$$

Un proces *Academie* fără blocaj este

$$ACADEMIE_NOUĂ=(ACADEMIE||VALET_0)$$

Istoria mesei celor cinci filozofi se datorește lui E. W. Dijkstra, iar introducerea valetului lui Carel S. Scholten.

2.5.4 Demonstrarea lipsei blocajului

În procesul *ACADEMIE* riscul blocajului era mai mult ca evident, de aceea pretenția că noul proces *ACADEMIE_NOUĂ* este lipsit de blocaj trebuie demonstrat cu prudență. Ceea ce trebuie să arătăm poate fi formulat

$$(ACADEMIE_NOUĂ/s)\neq STOP$$

pentru toate $s\in urme(ACADEMIE_NOUĂ)$

Demonstrația începe cu considerarea unei urme arbitrare *s* iar apoi arătăm că în toate cazurile există cel puțin un eveniment prin care *s* poate fi extinsă și totuși rămâne în *urme(ACADEMIE_NOUĂ)*. Întâi definim numărul filozofilor așezați

$așezați(s)=\#(s \upharpoonright A)-\#(s \upharpoonright R)$ unde *R* și *A* sunt definiți mai sus

Din cauză că (2.3.3 L1) $s \upharpoonright (R \cup A) \in urme(VALET_0)$ deducem că

$așezați(s) \leq 4$

Dacă am avea $așezați(s) \leq 3$, cel puțin un filozof s-ar mai putea așeza, deci nu este blocaj. În cazul $așezați(s)=4$, considerăm numărul filozofilor care mănâncă (au ambele furculițe ridicate). Dacă numărul lor este diferit de zero atunci un filozof care a mâncat pune furculița din stânga sa jos. Dacă nici un filozof nu mănâncă, să considerăm numărul furculițelor ridicate. Dacă acesta este mai mic sau egal cu trei, unul din filozofii care sunt așezați poate ridica furculița din stânga. Dacă sunt patru furculițe ridicate, atunci filozoful din stânga locului vacant poate ridica și furculița din dreapta. Dacă sunt cinci furculițe ridicate, cel puțin unul din filozofi mănâncă (nu avem blocaj !).

Demonstrația de mai sus implică analiza numărului de cazuri descrise formal în termenii comportării exemplului nostru. Să considerăm altă metodă de demonstrare. Să programăm un calculator pentru a găsi toate cazurile de blocaj. În general, nu putem ști dacă un program a verificat toate posibilitățile de blocaj. Dar în cazul unui sistem cu un număr finit de stări ca procesul *ACADEMIE* este suficient să considerăm acele urme a căror lungime nu depășesc limita numărului de stări. În general, numărul de stări a lui $(P \| Q)$ nu depășește produsul numărului de stări ale lui *P* înmulțit cu numărul de stări ale lui *Q*. Deoarece fiecare filozof are 6 stări și fiecare furculiță are 3 stări, numărul total de stări pentru *ACADEMIE* nu depășește

$6^5 \times 3^5 \approx$ 1,8 milioane

Deoarece alfabetul procesului *VALET* este conținut în acela al procesului *ACADEMIE*, atunci procesul *ACADEMIE_NOUĂ* nu are un număr mai mare de stări. Astfel, în aproape fiecare stare sunt posibile două sau mai multe evenimente, numărul urmelor care pot fi examinate în problema noastră depășind $2^{1,8 \text{ milioane}}$. Este clar că nici un calculator nu va putea explora toate aceste posibilități. De aceea, demonstrarea lipsei blocajului chiar pentru procese finite simple va rămâne responsabilitatea proiectantului de sisteme concurente.

2.5.5 Preluare infinită

În afară de pericolul blocajului mai există un pericol pentru sistemul analizat – acela ca un filozof care intenționează să mănânce să fie luat prin surpridere de un vecin. Să presupunem că în stânga unuia din filozofi există un vecin destul de lacom și că primul are de asemenea o mână stângă mai nesigură (de aceea mișcarea cu ea este încetinită). Astfel, înainte ca el să-și ridice furculița lui stângă, vecinul lacom (care este și foate îndemânatic) se grăbește, se așează, acaparează ambele furculițe și petrece un timp destul de lung mâncând. Apoi, el eliberează ambele furculițe și pleacă. Dar când flămânzește din nou, se grăbește la masă, se așează, ridică furculițele, în timp ce vecinul său nu mai apucă să ridice și el furculița comună. Deoarece acest ciclu se poate repeta la infinit, unul din filozofii așezați poate rămâne nemâncat.

Soluția corectă la această situație este să o privim ca o chestiune insolubilă, deoarece în situația că unul din filozofi se comportă ca cel lacom atunci un alt filozof va rămâne flămând o perioadă mai mare decât în mod obișnuit. Astfel se observă că nu există soluție care să aducă satisfacție totală, rezolvarea sigură fiind de a cumpăra mai multe furculițe și mai multe spaghetti.

Totuși este important de garantat că un filozof așezat va mânca, ceea ce implică modificarea comportării valetului. Ajutând un filozof să se așeze, el așteaptă până când acesta ridică ambele furculițe după care ajută un alt vecin să se așeze. Dar rămâne situația când unul din filozofi poate fi luat prin surprindere mereu. Să presupunem că valetul are o antipatie nejustificată față de unul din filozofi și permanent întârzie acțiunea de a-l conduce la scaun, chiar dacă filozoful este pregătit pentru acest eveniment. Aceasta este o altă posibilitate care nu a fost luată în considerare în exemplul noastru deoarece nu o putem distinge de situația când un filozof însuși așteaptă un timp foarte lung până flămânzește. Avem de a face cu o problemă de sincronizare pe care am hotărât să o ignorăm deliberat sau rămâne în sarcina unei alte faze a proiectării și implementării. Este sarcina celui care implementează să se asigure că orice eveniment ce devine posibil va avea loc într-un interval de timp rezonabil. De asemenea, implementarea în limbaje de nivel înalt nu trebuie să introducă asemenea întârzieri arbitrare în execuția programului, chiar dacă programatorul nu are mijloace de a impune sau descrie acest lucru.

2.6 Schimbare de simbol

Exemplul din paragraful precedent a făcut apel la două mulțimi de procese : filozofii și furculițele. În cadrul fiecărei mulțimi procesele au comportări foarte apropiate cu excepția numelor evenimentelor în care se angajează. În acest paragraf vom introduce o metodă potrivită pentru definirea unor procese cu

comportări similare. Fie f o injecţie (unu la unu) ce realizează corespondenţa între alfabetul lui P şi mulţimea de simboluri A

$$f{:}\alpha P \rightarrow A$$

Definim procesul $f(P)$ ca unul care se angajează în evenimentul $f(c)$ ori de câte ori P se angajează în c. Avem

$$\alpha f(P) = f(\alpha P)$$
$$urme(f(P)) = \{f^*(s) \mid s \in urme(P)\}$$

(Pentru definirea lui f^* vezi 1.9.1.).

Exemple

X1 În timp, preţurile produselor, după cum prea bine ştim, cresc. Pentru a reprezenta efectul inflaţiei, definim o funcţie f prin următoarele ecuaţii

$f(in2p)=in5p$ $\qquad f(mare)=mare$
$f(in1p)=in5p$ $\qquad f(mic)=mic$
$f(ex1p)=ex5p$

Noul automat de vânzare complex este

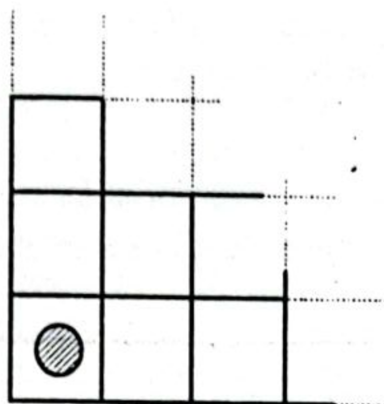
$AVCNOU=f(AVC)$ □

X2 Un obiect se comportă ca MM_0 (1.1.4 X2), dar efectuează acţiunile *stânga*, *dreapta* în loc de *sus*, *jos*

$f(sus)=dreapta$, $f(jos)=stânga$, $f(pe_loc)=pe_loc$,
$SD_0=f(MM_0)$ □

Principalul scop al tehnicii de schimbare a numelor evenimentelor este de a permite o compunere mai eficientă a proceselor în construcţii concurente.

X3 Un marker se mişcă pe o suprafaţă riglată infinită *stânga*, *dreapta*, *sus*, *jos*, margini existând doar în partea stângă şi în cea de jos



Markerul pornește din colțul din stânga jos. Ca și în 2.3 X2, mișcările pe verticală și orizontală pot fi modelate prin acțiuni independente ale unor procese separate. Numai în colțul din stânga jos markerul se poate angaja în evenimentul *pe_loc*, care însă necesită participarea simultană a celor două procese separate

$$SDSJ = SD_0 \| MM_0$$ □

X4 Vrem să conectăm două instanțe ale lui *COPIEBIT* (1.1.3 X7), astfel că fiecare bit emis de primul să fie primit de al doilea. Mai întâi trebuie să modificăm numele evenimentelor folosite pentru comunicația internă, așa încât vom introduce două noi evenimente *mij*.0 și *mij*.1 și definim funcțiile *f* și *g* pentru a schimba emiterea unuia din procese și primirea celuilalt

$f(emi.0) = g(pri.0) = mij.0$
$f(emi.1) = g(pri.1) = mij.1$
$f(pri.0) = pri.0, f(pri.1) = pri.1$
$g(emi.0) = emi.0, g(emi.1) = emi.1$

Rezultatul pe care-l așteptăm este

$$LANȚ2 = f(COPIEBIT) \| g(COPIEBIT)$$

De obsevat că fiecare emitere a unui 0 sau 1 de către operandul stâng al operatorului || este (prin definiția lui *f* și *g*) același eveniment (*mij*.0, *mij*.1) ca cel primit drept 0 sau 1 de către operandul din dreapta. Astfel este modelată comunicarea sincronizată a unor biți printr-un canal ce leagă două procese, ca în fig. 2.7

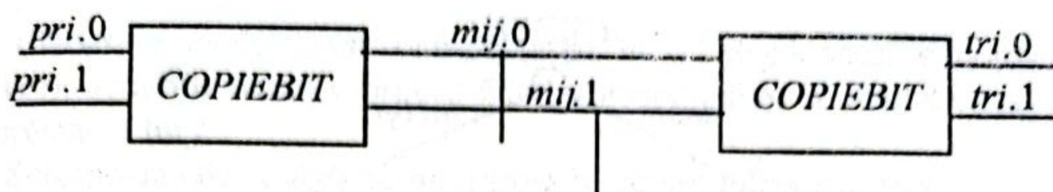


Figura 2.7

Operandul din stânga nu are de ales în ce privește valoarea emisă pe canalul de legătură, în timp ce operandul din dreapta este pregătit să se angajeze în oricare din evenimentele *mij*.0, *mij*.1. De aceea revine procesului care emite să determine cu fiecare ocazie care dintre cele două evenimente va apărea. Această metodă de comunicație între procese concurente va fi generalizată în cap. 4.

Să remarcăm că evenimentele *mij*.0 și *mij*.1 sunt în alfabetul procesului compus și pot fi observate (chiar controlate) de mediu. Cineva ar putea avea intenția de a ignora sau masca astfel de evenimente interne, dar în general mascarea ar putea duce la nedeterminism, astfel că problema este amânată pentru paragraful 3.5 □

X5 Vrem să reprezentăm comportarea unei variabile booleene dintr-un program oarecare. Evenimentele din alfabetul său sunt

*asign*0	asignează variabilei valoarea 0
*asign*1	asignează variabilei valoarea 1
*acces*0	accesează valoarea variabilei când este 0
*acces*1	accesează valoarea variabilei când este 1

Comportarea variabilei este foarte asemănătoare cu cea a automatului de băuturi (1.1.4 X1), astfel că definim

$BOOL = f(AK)$

unde definirea lui *f* este o chestiune banală. Se observă că nu se poate accesa o valoare până ce nu s-a asignat una. Încercarea de a accesa o valoare neasignată duce la blocaj - una din erorile subtile de programare. □

Arborescența lui *f*(*P*) poate fi construită din cea a lui *P* aplicând simplu funcția *f* etichetelor tuturor ramurilor. Deoarece *f* este o funcție injectivă, transformarea păstrează structura de arbore și distincția între etichetele arcelor pornind din același nod. De exemplu, reprezentarea lui *AVCNOU* este dată în fig. 2.8

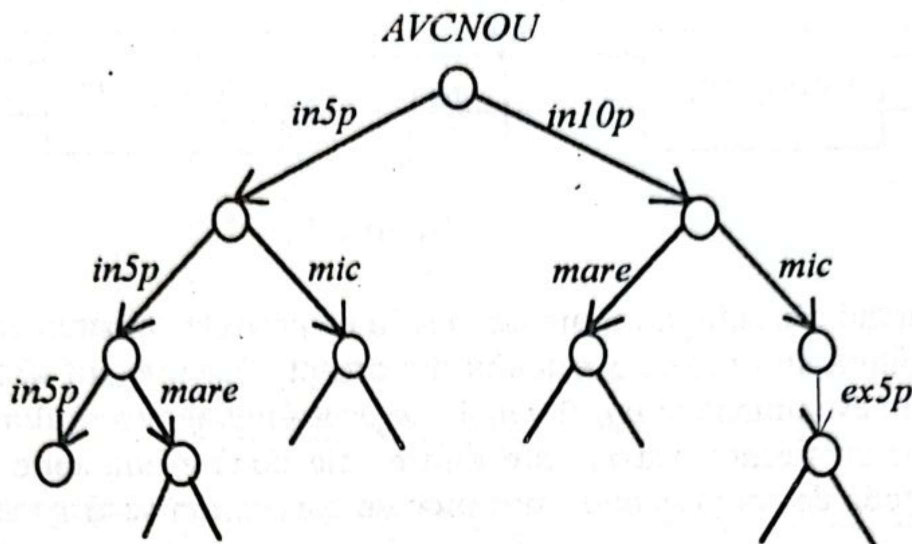


Figura 2.8

2.6.1 Legi

Schimbarea de simbol prin aplicarea unei funcții injective nu schimbă structura comportării unui proces. Aceasta rezultă din distributivitatea aplicării unei funcții față de alți operatori, așa cum rezultă din următoarele legi. Vom folosi următoarele definiții auxialiare

$f(B)=\{f(x) \mid x \in B\}$
f^{-1} inversa lui f
$f \circ g$ compunerea lui f cu g
f^{*} definită în paragraful 1.9.1

(nevoia de f^{-1} în următoarele legi este un motiv important de a insista ca f să fie injectivă)

După schimbarea de simbol, *STOP* tot nu se angajează în nici un eveniment din alfabetul său schimbat

L1 $f(STOP_A)=STOP_{f(A)}$

În cazul unei alegeri, atât simbolurile oferite pentru selecție cât și comportarea ulterioară sunt schimbate

L2 $f(x{:}B \rightarrow P(x))=(y{:}f(B) \rightarrow f(P(f^{-1}(y))))$

Utilizarea lui f^{-1} din partea dreaptă ne obligă la următoarele explicații. Să ne amintim că P reprezintă o funcție-proces ce are o comportare ulterioară depinzând de un x din mulțimea B. Dar variabila y din partea dreaptă ia valori din mulțimea $f(B)$. Astfel evenimentul corespunzător pentru P este $f^{-1}(y)$, care este

în B (deoarece $y \in f(B)$) iar comportarea lui P după acest eveniment este $P(f^{-1}(y))$, acţiunile acestui proces fiind corespunzător schimbate mai departe prin aplicarea lui f.

Schimbarea de simbol se distribuie în compunerea paralelă

L3 $f(P\|Q)=f(P)\|f(Q)$

Schimbarea de simbol se distribuie într-un mod ceva mai complex în recursivitate, alfabetul modificându-se adecvat

L4 $f(\mu X{:}A.F(X))=(\mu Y{:}f(A).f(F(f^{-1}(Y))))$

Din nou, utilizarea lui f^{-1} în partea dreaptă necesită unele explicaţii. Să ne amintim că o condiţie pentru validitatea recursivităţii din partea stângă cere ca F să fie o funcţie cu argument un proces ce are alfabetul A, generând un proces cu acelaşi alfabet. În partea dreaptă, Y este o variabilă modelând procesele cu alfabetul $f(A)$ şi nu poate fi folosită ca argument al lui F, alfabetul său trebuind să fie mulţimea A. Aceasta se realizează aplicând funcţia inversă f^{-1} variabilei Y. Astfel $F(f^{-1}(Y))$ are alfabetul A, astfel că aplicarea lui f va transforma alfabetul la $f(A)$, asigurând validitatea recursivităţii şi în partea dreaptă a legii.

Compunerea a două schimbări de simbol este definită prin compunerea a două funcţii schimbare de simbol

L5 $f(g(P))=(f \circ g)(P)$

Urmele unui proces după schimbarea de simbol se obţin simplu prin schimbarea simbolilor individuali în orice urmă a procesului original

L6 $urme(f(P))=\{f^*(s) \mid s \in urme(P)\}$

Explicaţia următoarei şi ultimei legi este similară cu aceea a lui L6

L7 $f(P)/f^*(s)=f(P/s)$

2.6.2 Etichetarea proceselor

Schimbarea de simbol este în special utilă în construirea claselor de procese similare concurente, relativ la acţiunile faţă de mediu, dar care nu interacţionează de loc între ele. Aceasta înseamnă că procesele trebuie să aibă alfabete disjuncte mutual. Pentru a realiza aceasta, fiecare proces este etichetat diferit, fiecare eveniment al său având aceeaşi etichetă ca şi procesul. Un

eveniment etichetat va fi o pereche $l.x$, unde l este eticheta iar x este un simbol corespunzător unui eveniment. Un proces P etichetat cu l este notat

$l{:}P$

Acest proces se angajează în evenimentul $l.x$ ori de câte ori P se angajează în x. Evident etichetarea poate fi definită cu ajutorul unei funcții

$f_l(x)=l.x$ pentru toți x din αP
$l{:}P=f_l(P)$

Exemple

X1 O pereche de automate care stau unul lângă altul

(stânga:AVS)||(dreapta:AVS)

Alfabetele celor două procese sunt disjuncte și orice eveniment care apare are eticheta automatului unde se produce. Dacă automatele nu ar fi fost etichetate înainte de a fi fost așezate, acționând în paralel, orice eveniment ar fi necesitat participarea ambelor mașini. Astfel perechea nu s-ar fi deosebit de un singur automat. Aceasta este o consecință a faptului că

(AVS||AVS)=AVS □

Etichetarea proceselor permite folosirea lor ca și variabilele declarate local unei proceduri într-un limbaj de nivel înalt.

X2 Comportarea unei variabile boolene este modelată prin procesul *VARB* (2.6 X5). Comportarea unei proceduri este descrisă de procesul *UTIL*, care asignează și încarcă valorile a două variabile boolene numite b și c. Astfel $\alpha UTIL$ include evenimente compuse ca

*b.asign*0 asignează valoarea 0 lui b
*c.acces*1 accesează valoarea curentă a lui c când este 1.

Procesul *UTIL* interacționează în paralel cu cele două variabile boolene

b:VARB||c:VARB||UTIL

În procedura *UTIL* se pot realiza următoarele acțiuni

b:=false;P prin $(b.asign0 \rightarrow P)$

$b := \neg c; P$ prin $(c.acces0 \rightarrow b.asign1 \rightarrow P$
$| c.acces1 \rightarrow b.asign0 \rightarrow P)$

De observat cum valoarea curentă a variabilei este identificată prin alegerea între *acces*0 și *acces*1. Această alternativă afectează într-un mod adecvat comportarea ulterioară a lui *UTIL*. □

În exemplul X2 și în următoarele ar fi mult mai comod să definim efectul unei singure asignări, de exemplu

$b := false$

decât perechea

$b := false; P$

care menționează explicit restul de program din *P*. Semnificația perechii din urmă va fi introdusă în cap. 5.

X3 Un proces *UTIL* prelucrează două variabile contor etichetate cu *l* și *m*. Ele sunt inițializate cu 0 și respectiv 3. Procesul *UTIL* le incrementează folosind evenimentele *l.sus* sau *m.sus* și le decrementează (când sunt pozitive) prin *l.jos* sau *m.jos*. Un test de zero este asigurat de evenimentele *l.pe_loc* și *m.pe_loc*. Astfel, pentru variabilele *l* și *m* poate fi folosit procesul MM_n (1.1.4 X2) după o etichetare adecvată

$(l{:}MM_0 \| m{:}MM_3 \| UTIL)$

În interiorul procesului *UTIL* se realizează următoarele efecte (exprimate convențional)

$(m := m+1; P)$ prin $(m.sus \rightarrow P$
if $l=0$ **then** P **else** Q prin $(l.pe_loc \rightarrow P | l.jos \rightarrow l.sus \rightarrow Q)$

Să remarcăm cum se realizează testul pentru zero : există o alegere între evenimentul *l.jos*, pentru decrementarea contorului cu unu și evenimentul *l.pe_loc*. Contorul etichetat cu *l* selectează între aceste două evenimente : dacă valoarea variabilei este zero este selectat *l.pe_loc* iar dacă este diferită de zero, cealaltă alternativă. Dar în al 2-lea caz valoarea variabilei a fost decrementată și restabilirea ei la valoarea originală se face prin *l.sus*. În următorul exemplu restabilirea valorii originale este mai complexă.

$(m := m+1; P)$ este implementată prin *ADUN*

unde *ADUN* este definit recursiv

$ADUN = JOS_0$
şi $JOS_i = (l.sos \rightarrow JOS_{i+1} | l.pe_loc \rightarrow SUS_i)$
şi $SUS_0 = P$
şi $SUS_{i+1} = l.sus \rightarrow m.sus \rightarrow SUS_i$

Procesele JOS_i testează valoarea iniţială a lui *l* decrementând-o până la zero. Procesele SUS_i adună valoarea găsită atât la *m* cât şi la *l*, refăcându-l astfel pe *l* la valoarea iniţială şi adunându-l practic pe *l* la *m*. □

Efectul unui masiv de variabile poate fi realizat printr-o colecţie de procese concurente, fiecare etichetat cu un index din masiv.

X4 Scopul procesului *TEST* este de a înregistra dacă evenimentul *ap* a apărut sau nu. La prima apariţie a lui *ap* răspunde *nu* şi la fiecare din apariţiile următoare răspunde *da*

$\alpha TEST = \{ap, nu, da\}$
$TEST = ap \rightarrow nu \rightarrow \mu X.(ap \rightarrow da \rightarrow X)$

Acest proces poate fi folosit pentru a simula comportarea unui masiv cu elemente întregi

$INT3 = (0{:}TEST) || (1{:}TEST) || (2{:}TEST) || (3{:}TEST)$

Întregul masiv poate fi iarăşi etichetat înainte de utilizare

$m{:}INT3 || UTIL$

Fiecare eveniment în $\alpha(m{:}INT3)$ este un triplet, de exemplu *m.2.ap*. În procesul *UTIL*, efectul pentru

if $2 \in m$ **then** *P* **else** $(m := m \cup \{2\}; Q)$

poate fi realizat prin

$m.2.ap \rightarrow (m.2.da \rightarrow P | m.2.nu \rightarrow Q)$ □

2.6.3 Implementare

În general, pentru a implementa schimbarea de simbol trebuie să ştim inversa *g* a funcţiei *f*. Trebuie de asemenea să ne asigurăm că *g* va da un răspuns spe-

cial "*BLIP* când este aplicată la un argument din afara domeniului lui *f*. Implementarea este bazată pe 2.6.1 L4.

> *schimb*(*g*,*P*)=λ*x*. **if** *g*(*x*)="*BLIP* **then** "*BLIP*
> **else if** *P*(*g*(*x*))="*BLIP* **then** "*BLIP*
> **else** *schimb*(*g*,*P*(*g*(*x*)))

Cazul special al etichetării procesului poate fi implementat mult mai simplu. Evenimentul compus *l.x* este reprezentat ca o pereche de atomi *cons*("*l*,"*x*). Deci (*l*:*P*) este implementat de

> *etich*(*l*,*P*)=λ*y*. **if** *null*(*y*)∨*atom*(*y*) **then** "*BLIP*
> **else if** *car*(*y*)≠l **then** "*BLIP*
> **else if** *P*(*cdr*(*y*))="*BLIP* **then** "*BLIP*
> **else** *etich*(*l*,*P*(*cdr*(*y*)))

2.6.4 Etichetare multiplă

Definiția etichetării poate fi extinsă pentru a permite fiecărui eveniment să ia orice etichetă *l* dintr-o mulțime *L*. Dacă *P* este un proces, (*L*:*P*) este definit ca un proces care se comportă exact ca *P*, cu excepția faptului că se angajează în evenimentul *l.c* (unde $l \in L$ și $c \in \alpha P$) ori de câte ori *P* s-ar angaja în *c*. Alegerea unei etichete *l* este făcută independent, cu orice ocazie, de mediul lui (*L*:*P*).

Exemple

X1 Relativ la problema mesei celor cinci filozofi, introducem un valet tânăr care își ajută numai stăpânul său înspre și de la scaun și stă în spatele acestui scaun când acesta mănâncă

> α*VAL_T*={*așează*,*ridică*}
> *VAL_T*=(*așează*→*ridică*→*VAL_T*)

Acest proces poate să-și împartă serviciile la cinci stăpâni (dar servind numai pe unul la un moment dat)

> *E*={0,1,2,3,4}
> *VAL_T_M*=(*E*:*VAL_T*)

Valetul tânăr multiplu ar putea fi angajat pentru a proteja de înfometare filozofii care iau masa când valetul (2.5.3.) este în vacanță. Desigur, filozofii

ar putea deveni mai flămânzi în vacanță deoarece numai unul dintre ei este condus la masă la un moment dat. □

Dacă E conține mai mult decât o etichetă, imaginea arborescentă a lui E:P este similară cu aceea pentru P dar mai stufoasă, în sensul că sunt mult mai multe ramuri plecând din același nod. De exemplu, imaginea lui VAL_T este un trunchi fără ramuri (fig. 2.9). Totuși imaginea lui $\{0,1\}$:VAL_T este un arbore binar complet (fig. 2.10). Arborele pentru VAL_T_M este mult mai stufos.

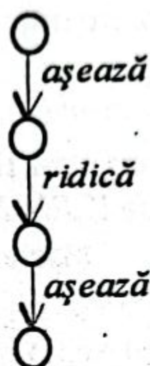


Figura 2.9

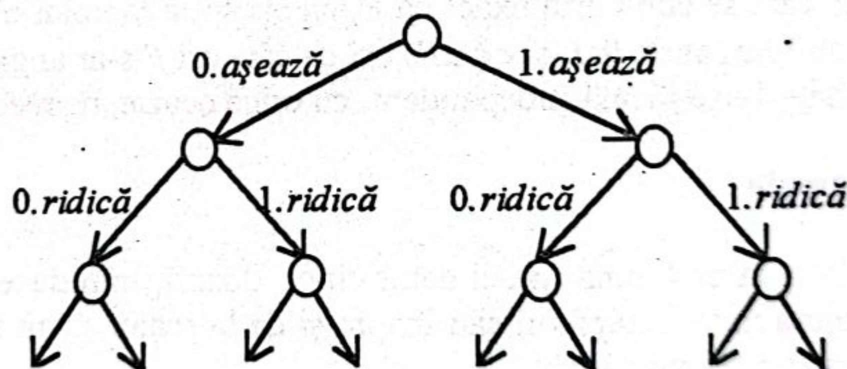


Figura 2.10

În general, etichetarea multiplă este utilă când folosim serviciile unui singur proces într-un număr de alte procese etichetate, admițând că mulțimea etichetelor este cunoscută dinainte. Această tehnică va fi exploatată mai intens în cap. 6.

2.7 Specificații

Fie P și Q procese care doresc să activeze concurent și să presupunem că am demonstrat

P **sat** $S(ur)$
și Q **sat** $T(ur)$

Fie ur o urmă a lui $(P\|Q)$. Din 2.3.3 L1 avem că $(ur\upharpoonright\alpha P)$ este o urmă a lui P și deci satisface S

$$S(ur\upharpoonright\alpha P)$$

Similar, $(ur\upharpoonright\alpha Q)$ este urmă a lui Q așa că

$$T(ur\upharpoonright\alpha Q)$$

Acest argument este valabil pentru orice urmă a lui $(P\|Q)$. De aceea putem deduce

$$(P\|Q) \text{ sat } (S(ur\upharpoonright\alpha P)\wedge T(ur\upharpoonright\alpha Q))$$

Acest raționament formal este rezumat în legea

L1 Dacă P **sat** $S(ur)$
și Q **sat** $T(ur)$
atunci $(P\|Q)$ **sat** $(S(ur\upharpoonright\alpha P)\wedge T(ur\upharpoonright\alpha Q))$

Exemplu

X1 (Vezi 2.3.1. X1)

Fie $\alpha P=\{a,c\}$, $\alpha Q=\{b,c\}$
Fie $P=(a\rightarrow c\rightarrow P)$
Fie $Q=(c\rightarrow b\rightarrow Q)$

Vrem să dovedim că

$$(P\|Q) \text{ sat } 0\leq ur\downarrow a-ur\downarrow b\leq 2$$

Demonstrația lui 1.10.2 X2 poate fi evident adaptată pentru a arăta că

P **sat** $(0\leq ur\downarrow a-ur\downarrow c\leq 1)$
și Q **sat** $(0\leq ur\downarrow c-ur\downarrow b\leq 1)$

Din L1 rezultă că

$(P||Q)$ **sat** $(0 \leq (ur \upharpoonright \alpha P) \downarrow a - (ur \upharpoonright \alpha P) \downarrow c \leq 1 \wedge 0 \leq (ur \upharpoonright \alpha Q) \downarrow c - (ur \upharpoonright \alpha Q) \downarrow b \leq 1)$
$\Rightarrow 0 \leq ur \downarrow a - ur \downarrow b \leq 2$ deoarece $(ur \upharpoonright A) \downarrow a = ur \downarrow a$ ori de câte ori $a \in A$

□

Deoarece legile pentru sat permit lui *STOP* să îndeplinească orice specificație, raționamentul bazat pe aceste legi nu poate niciodată dovedi absența blocajului, legi mai puternice urmând a fi date în paragraful 3.7. Pe moment, o cale de a elimina riscul blocării este o demonstrație atentă ca în paragraful 2.5.4. O altă metodă este de a arăta că procesul definit cu operatorul pentru concurență este echivalent cu un proces fără blocaj definit fără acest operator, așa cum a fost făcut în 2.3.1 X1. Totuși asemenea demonstrații presupun transformări lungi și anevoioase. Ori de câte ori este posibil trebuie apelat la legi generale ca

L2 Dacă P și Q nu se opresc (blochează) niciodată și dacă $(\alpha P \cap \alpha Q)$ conține cel puțin un eveniment, atunci $(P||Q)$ definit în X1 nu se va opri niciodată.

Exemplu

X2 Procesul $(P||Q)$ definit în X1 nu se va opri niciodată din cauză că

$\alpha P \cap \alpha Q = \{c\}$. □

Legea pentru schimbare de simbol este

L3 Dacă P **sat** $S(ur)$
atunci $f(P)$ **sat** $S(f^{-1*}(ur))$

Utilizarea lui f^{-1*} în această lege necesită explicații. Fie *ur* o urmă a lui $f(P)$. Atunci $f^{-1*}(ur)$ este o urmă a lui P. Ipoteza lui L3 afirmă că orice urmă a lui P satisface S. Urmează că $f^{-1*}(ur)$ satisface S, ceea ce este exact concluzia lui L3.

2.8 Teoria matematică a proceselor deterministe

În descrierea proceselor am enunțat un mare număr de legi și ocazional le-am folosit în demonstrații. Legile au fost justificate (în cele mai multe din cazuri) prin raționamente elementare în raport de ce ne așteptam și doream să fie adevărat. Pentru un cititor cu reflexe de matematică aplicată sau inginerie aceasta poate fi de ajuns. Dar întrebarea care se naște este limpede : sunt aceste legi adevărate de fapt ? Sunt ele consistente ? Ar trebui să fie mai multe ? Sau sunt complete în sensul că ele permit ca toate problemele legate de

procese să fie dovedite pe baza lor ? S-ar putea cineva descurca cu mai puţine legi sau mai simple ? Acestea sunt întrebări la care se poate răspunde printr-o investigaţie matematică mai profundă.

2.8.1 Definiţii de bază

În construirea unui model matematic al unui sistem fizic o metodă bună este de a defini conceptele de bază în termenii proprietăţilor care pot fi direct sau indirect observate sau măsurate. Pentru un proces determinist P suntem familiari cu două asemenea proprietăţi

αP	mulţimea evenimentelor în care procesul este evident capabil de a se angaja
urme(P)	mulţimea secvenţelor de evenimente în care procesul poate participa dacă este nevoie.

Am explicat cum aceste două mulţimi trebuie să satisfacă cele trei legi din 1.8.1, L6, L7, L8. Considerăm acum o pereche arbitrară de mulţimi (A,S) care satisface aceste trei legi. Perechea identifică unic un proces P a cărui urme S sunt construite potrivit următoarele definiţii

Fie $P^0=\{x \mid \langle x\rangle \in S\}$

şi $P(x)$ procesul ale cărui urme sunt $\{t \mid \langle x\rangle ^\frown t \in S\}$ pentru toţi x în P^0

Atunci $\alpha P=A$

şi $P=(x{:}P^0 \rightarrow P(x))$

Mai mult, perechea (A,S) poate fi formată din ecuaţiile

$A=\alpha P$

$S=urme(x{:}P^0 \rightarrow P(x))$

Din această cauză există o corespondenţă biunivocă între fiecare proces P şi perechea de mulţimi $(\alpha P, urme(P))$. În matematică aceasta este o justificare suficientă pentru echivalarea celor două concepte.

D0 Un proces determinist este o pereche

(A,S)

unde A este orice mulţime de simboluri şi S este orice submulţime a lui A^* care satisface cele două condiţii

C0 $\langle\rangle \in S$
C1 $\forall s,t.\ s\frown t \in S \Rightarrow s \in S$

Cel mai simplu exemplu de proces care îndeplineşte această definiţie este acela care nu face nimic

D1 $STOP_A = (A, \{\langle\rangle\})$

La cealaltă extremă este procesul care activează continuu

D2 $RUN_A = (A, A^*)$

Diferiţii operatori de la procese pot fi acum definiţi formal arătând cum alfabetul şi urmele rezultatului provin din alfabetul şi urmele operanzilor.

D3 $(x{:}B \rightarrow (A, S(x))) = (A, \{\langle\rangle\} \cup \{\langle x\rangle \frown s \mid x \in B \wedge s \in S(x)\})$ dacă $B \subseteq A$
D4 $(A,S)/s = (A, \{t \mid (s\frown t) \in S\})$ dacă $s \in S$

D5 $\mu X{:}A.F(X) = (A, \bigcup_{n\geq 0} urme(F^n(STOP_A)))$
D6 $(A,S) \| (B,T) = (A \cup B,\ \{s \mid s \in (A \cup B)^* \wedge (s \upharpoonright A) \in S \wedge (s \upharpoonright B) \in T\})$
D7 $f(A,S) = (f(A),\ \{f^*(s) \mid s \in S\})$ dacă f este injectivă

Desigur, este necesar să dovedim că în partea dreaptă a acestor definiţii sunt de fapt procese, cu alte cuvinte satisfac condiţiile C0 şi C1 din D0. Din fericire aceasta este uşor de arătat.

În cap. 3 se va vedea că D0 nu este o definiţie adecvată a conceptului de proces din cauză că nu cuprinde posibilitatea de nedeterminism. De aceea, este necesară o definiţie mai generală şi mai complicată. Toate legile pentru procese nedeterministe sunt adevărate pentru cele deterministe dar procesele deterministe au câteva legi în plus, de exemplu

$$P \| P = P$$

Pentru a evita confuzia, în această carte am evitat marcarea unor astfel de legi, astfel că toate legile pot fi folosite în siguranţă la procese nedeterministe ca şi la cele deterministe (cu excepţia 2.2.1 L3A, 2.2.3 L1, 2.3.1 L3A, 2.3.3 L1, L2,L3, care sunt false pentru procese conţinând *CHAOS* (3.8)).

2.8.2 Teoria de punct fix

Obiectivul acestui paragraf este de a contura demonstraţia teoremei fundamentale a recursivităţii, astfel ca un proces definit recursiv (2.8.1 D5) să fie soluţie a ecuaţiei sale recursive

$$\mu X.F(X) = F(\mu X.F(X))$$

Metoda urmează teoria de punct fix a lui Scott. Mai întâi trebuie să stabilim o relaţie de ordine între procese $\sqsubseteq$

D1 $(A,S) \sqsubseteq (B,T) = (A=B \wedge S \subseteq T)$

Două procese sunt comparabile în această ordine dacă au acelaşi alfabet şi unul face tot ce face celălalt şi chiar mai mult. Această ordonare este parţială în sensul că

L1 $P \sqsubseteq P$

L2 $P \sqsubseteq Q \wedge Q \sqsubseteq P \Rightarrow P = Q$

L3 $P \sqsubseteq Q \wedge Q \sqsubseteq R \Rightarrow P \sqsubseteq R$

Un lanţ în ordinea parţială este o secvenţă infinită de elemente

$$\{P_0, P_1, P_2, \ldots\}$$

astfel că $P_i \leq P_{i+1}$ pentru toţi i

Definim limita (cea mai mică margine superioară) unui astfel de lanţ

$$\bigsqcup_{i \geq 0} P_i = (\alpha P_0, \bigcup_{i \geq 0} urme(P_i))$$

Pe viitor vom aplica operatorul de limită $\bigsqcup$ numai proceselor care formează un lanţ.

O ordine parţială se spune că este completă dacă are un cel mai mic element şi toate lanţurile au o cea mai mică margine superioară. Mulţimea tuturor proceselor cu un alfabet A dat formează o relaţie de ordine parţială completă (o.p.c) dacă satisface legile

L4 $STOP_A \sqsubseteq P$ dacă $\alpha P = A$

L5 $P_i \sqsubseteq \bigsqcup_{i \geq 0} P_i$

L6 $(\forall i\geq 0.P_i\sqsubseteq Q)\Rightarrow(\bigsqcup_{i\geq 0} P_i)\sqsubseteq Q$

Mai mult, definiția lui μ (2.8.1 D5) poate fi reformulată în termeni de limită

L7 $\mu X{:}A.F(X)=(\bigsqcup_{i\geq 0} F^i(STOP_A))$

O funcție F de la o opc la alta (sau aceeași) se spune că este continuă dacă se distribuie limitelor tuturor șirurilor

$$F(\bigsqcup_{i\geq 0} P_i)=\bigsqcup_{i\geq 0} F(P_i) \qquad \text{dacă } \{P_i \mid i\geq 0\} \text{ este lanț}\}$$

(Toate funcțiile continue sunt monotone în sensul că

$$P\sqsubseteq Q\Rightarrow F(P)\sqsubseteq F(Q) \quad \text{pentru toți } P \text{ și } Q$$

astfel că membrul drept al ecuației anterioare este de asemenea limita unui șir crescător). O funcție G de mai multe argumente este definită drept continuă dacă este continuă în fiecare din argumente separat, de exemplu

$$G((\bigsqcup_{i\geq 0} P_i),Q)=\bigsqcup_{i\geq 0} G(P_i,Q) \qquad \text{pentru toți } Q$$

și $$G(Q,(\bigsqcup_{i\geq 0} P_i))=\bigsqcup_{i\geq 0} G(Q,P_i) \qquad \text{pentru toți } Q$$

Compunerea funcțiilor continue ne dă o funcție continuă. Într-adevăr, orice expresie construită prin aplicarea unor funcții continue la orice număr sau combinație a variabilelor este continuă în fiecare din acele variabile. De exemplu, dacă G, F și H sunt continue

$$G(F(X),H(X,Y))$$

este continuă în X deoarece

$$G(F(\bigsqcup_{i\geq 0} P_i),H((\bigsqcup_{i\geq 0} P_i),Y))=\bigsqcup_{i\geq 0} G(F(P_i),H(P_i,Y)) \qquad \text{pentru toți } Y$$

Toți operatorii (cu excepția lui /) definiți în D3 - D7 sunt continui în sensul definit mai sus

L8 $(x{:}B \to (\bigsqcup_{i\geq 0} P_i(x))) = \bigsqcup_{i\geq 0} (x{:}B \to P_i(x))$

L9 $\mu X{:}A.F(X,(\bigsqcup_{i\geq 0} P_i)) = \bigsqcup_{i\geq 0} \mu X{:}A.F(X,P_i)$ dacă F este continuă

L10 $(\bigsqcup_{i\geq 0} P_i) \| Q = Q \| (\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} (Q \| P_i)$

L11 $f(\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} f(P_i)$

Prin urmare dacă $F(X)$ este orice expresie construită numai în baza acestor operatori va fi continuă în X. Acum este posibil să demonstrăm teorema de punct fix

$$F(\mu X{:}A.F(X)) = F(\bigsqcup_{i\geq 0} F^i(STOP_A)) \quad \text{definiția lui } \mu$$

$$= \bigsqcup_{i\geq 0} F(F^i(STOP_A)) \quad \text{continuitatea lui } F$$

$$= \bigsqcup_{i\geq 1} F^i(STOP_A) \quad \text{definiția lui } F^{i+1}$$

$$= \bigsqcup_{i\geq 0} F^i(STOP_A) \quad STOP_A \sqsubseteq F(STOP_A)$$

$$= \mu X{:}A.F(X) \quad \text{definiția lui } \mu$$

Această demonstrație s-a bazat numai pe faptul că F este continuă. Existența unei gărzi a lui F este necesară pentru a stabili unicitatea soluției.

2.8.3 Unicitatea soluțiilor

În acest paragraf vom trata mai formal raționamentul dat în paragraful 1.1.2 pentru a arăta că o ecuație definind un proces printr-o recursivitate cu gardă are o singură soluție. Astfel, vom face mai explicite condițiile generale pentru unicitatea unor astfel de soluții. Pentru ușurință ne ocupăm de ecuații simple. Metoda se poate ușor extinde la sisteme de ecuații.

Dacă P este un proces și n este un număr natural, definim $(P \upharpoonright n)$ ca un proces care se comportă ca P pentru primele n evenimente și apoi se oprește. Formal

$(A,S)\upharpoonright n=(A, \{s \mid s\in S\wedge \#s\leq n\})$

Urmează că

L1 $P\upharpoonright 0=STOP$

L2 $P\upharpoonright n\sqsubseteq P\upharpoonright(n+1)\sqsubseteq P$

L3 $P=\bigsqcup_{n\geq 0} (P\upharpoonright n)$

L4 $\bigsqcup_{n\geq 0} P_n=\bigsqcup_{n\geq 0} (P_n\upharpoonright n)$

Fie F o funcție monotonă definită și cu valori în procese. F se spune că este constructivă dacă

$F(X)\upharpoonright(n+1)=F(X\upharpoonright n)\upharpoonright(n+1)$ pentru toți X

Aceasta înseamnă că comportarea lui $F(X)$ în primii $n+1$ pași este determinată numai de comportarea lui X în primii n pași. Astfel dacă $s\neq\langle\rangle$

$s\in urme(F(X))\equiv s\in urme(F(X\upharpoonright(\#s-1)))$

Prefixul este un prim exemplu de funcție constructivă, deoarece

$(c\rightarrow P)\upharpoonright(n+1)=(c\rightarrow(P\upharpoonright n))\upharpoonright(n+1)$

Alegerea generală este de asemenea constructivă

$(x{:}B\rightarrow P(x))\upharpoonright(n+1)=(x{:}B\rightarrow(P(x)\upharpoonright n))\upharpoonright(n+1)$

Funcția identitate I nu este constructivă deoarece

$$\begin{aligned}I\upharpoonright(c\rightarrow P)\upharpoonright 1&=c\rightarrow STOP\\&\neq STOP\\&=I((c\rightarrow P)\upharpoonright 0)\upharpoonright 1\end{aligned}$$

Putem acum formula teorema fundamentală

L5 Fie F o funcție constructivă. Ecuația

$X=F(X)$

are o singură soluție pentru X.

Demonstrație. Fie X o soluție arbitrară
Demonstrăm mai întâi prin inducție lema

$$X\upharpoonright n=F^n(STOP)\upharpoonright n$$

Avem $X\upharpoonright 0=STOP=STOP\upharpoonright 0=F^0(STOP)\upharpoonright 0$

Inducția

$$\begin{aligned} X\upharpoonright(n+1)&=F(X)\upharpoonright(n+1) && \text{deoarece } X=F(X)\\ &=F(X\upharpoonright n)\upharpoonright(n+1) && F \text{ este constructivă}\\ &=F(F^n(STOP)\upharpoonright n)\upharpoonright(n+1) && \text{ipoteză}\\ &=F(F^n(STOP))\upharpoonright(n+1) && F \text{ este constructivă}\\ &=F^{n+1}(STOP)\upharpoonright(n+1) && \text{definiția } F^n \end{aligned}$$

Acum revenim la teorema dată

$$X=\bigsqcup_{n\geq 0}(X\upharpoonright n) \qquad \text{L3}$$

$$=\bigsqcup_{n\geq 0}F^n(STOP)\upharpoonright n \qquad \text{demonstrația de mai sus}$$

$$=\bigsqcup_{n\geq 0}F^n(STOP) \qquad \text{L4}$$

$$=\mu X.(X) \qquad \text{2.8.2 L7}$$

Deci toate soluțiile lui $X=F(X)$ sunt egale cu $\mu X.F(X)$; cu alte cuvinte $\mu X.F(X)$ este singura soluție a ecuației. □

Utilitatea acestei teoreme este mult mai mare dacă putem recunoaște clar care funcții sunt constructive și care nu. Să definim o funcție nedestructivă G care satisface

$$G(P)\upharpoonright n=G(P\upharpoonright n)\upharpoonright n \qquad \text{pentru toți } n \text{ și } P$$

Transformarea alfabetului este nedestructivă în acest sens, deoarece

$$f(P)\upharpoonright n=f(P\upharpoonright n)\upharpoonright n$$

La fel este și funcția identitate. Orice funcție monotonă care este constructivă este de asemenea nedestructivă. Dar operatorul *după* este destructiv, deoarece

$$\begin{aligned}((c \rightarrow c \rightarrow STOP)/\langle c\rangle)\upharpoonright 1 &= c \rightarrow STOP\\ &\neq STOP\\ &= (c \rightarrow STOP)/\langle c\rangle\\ &= (((c \rightarrow c \rightarrow STOP)\upharpoonright 1)/\langle c\rangle)\upharpoonright 1\end{aligned}$$

Orice compunere de funcții nedestructive este de asemenea nedestructivă (G și H), din cauză că

$$G(H(P))\upharpoonright n = G(H(P)\upharpoonright n)\upharpoonright n = G(H(P\upharpoonright n)\upharpoonright n)\upharpoonright n = G(H(P\upharpoonright n))\upharpoonright n$$

Mai important, orice compunere a unei funcții constructive cu una nedestructivă este constructivă. Dacă F, G, ..., H sunt nedestructive și una dintre ele este constructivă atunci

$$F(G(\ldots(H(X))\ldots))$$

este o funcție constructivă în X.

Raționamentele de mai sus se extind ușor la funcții de mai multe argumente. De exemplu compunerea concurentă este nedestructivă (în ambele argumente) deoarece

$$(P \| Q)\upharpoonright n = ((P\upharpoonright n) \| (Q\upharpoonright n))\upharpoonright n$$

Fie E o expresie conținând variabila de proces X. Atunci E se spune că este cu gardă în X dacă orice apariție a lui X în E se face printr-o funcție constructivă aplicată variabilei și nu una destructivă. Astfel următoarea expresie este cu gardă în X

$$(c \rightarrow X \mid d \rightarrow f(X \| P) \mid c \rightarrow (f(x) \| Q)) \| ((d \rightarrow X) \| R)$$

Cea mai importantă consecință este că proprietatea de constructivitate poate fi definită sintactic din următoarele condiții pentru gardă

D0 O expresie construită numai cu ajutorul operatorilor concurență, schimbare de simbol și alegere generală se spune că este păstrătoare de gardă.

D1 O expresie care nu conține pe X se spune că este cu gardă în X.

D2 O alegere generală

$(x{:}B \rightarrow P(X,x))$

este cu gardă în X dacă $P(X, x)$ este păstrătoare de gardă pentru toți x.

D3 O schimbare de simbol $f(P(X))$ este cu gardă în X dacă $P(X)$ este cu gardă în X.

D4 Un sistem concurent $P(X)\|Q(X)$ este cu gardă în X dacă atât $P(X)$ cât și $Q(X)$ sunt cu gardă în X.

În final ajungem la concluzia

L6 Dacă E este cu gardă în X atunci ecuația

$X=E$

are soluție unică.

3 Nedeterminism

3.1 Introducere

Operatorul alegere $(x:B \rightarrow P(x))$ definește un proces care poate avea mai multe comportări posibile. Operatorul concurență || permite altui proces să facă o selecție între alternativele oferite de mulțimea *B*. De exemplu, automatul de dat rest *SCH5C* (1.1.3 X2) oferă clientului alegerea restului cu trei monede mici și una mare sau două mari și una mică. Astfel de procese se numesc *deterministe*, din cauză că ori de câte ori există mai mult de un eveniment, alegerea între ele este controlată din exterior, din mediul în care se află procesul. Alegerea este determinată atât în sensul că mediul o poate face el însuși sau în sensul mai slab că mediul poate observa care alegere a fost făcută chiar în momentul când se face ea.

Uneori un proces are o gamă de comportări posibile dar mediul procesului nu are nici o posibilitate să influențeze sau chiar să observe selecția între alegerile făcute. De exemplu, un alt automat de dat rest poate da rest în orice situație de mai sus, dar alegerea între ele nu poate fi controlată sau chiar prezisă de client. Alegerea este făcută intern de mașina însăși într-un mod arbitrar sau nedeterminist. Mediul nu poate controla alegerea sau măcar să o observe, nici nu poate să depisteze când a fost făcută alegerea cu toate că poate să-și dea seama mai târziu care alegere a fost făcută din comportarea ulterioară a procesului.

Nu este nimic misterios cu acest fel de nedeterminism. El provine dintr-o decizie deliberată de a ignora factorii care influențează selecția. De exemplu, combinația de rest dată de mașină poate depinde într-un fel de încărcarea cu monezi mari și mici, însă aceste evenimente au fost excluse din alfabet. Astfel nedeterminismul este util pentru a menține un nivel înalt de abstractizare în descrierile comportării sistemelor.

3.2 SAU nedeterminist

Dacă *P* și *Q* sunt procese, atunci introducem notația

$P \sqcap Q$ (P sau Q)

care înseamnă un proces ce se comportă atât ca P cât şi ca Q, unde selecţia între ele este făcută arbitrar fără cunoştinţa sau controlul mediului extern. Alfabetele operanzilor se presupun a fi aceleaşi.

$$\alpha(P \sqcap Q)=\alpha P=\alpha Q$$

Exemple

X1 Un automat de rest cu două combinaţii

$$SCH5D=(in5p \rightarrow (ex1p \rightarrow ex1p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5D) \sqcap (ex2p \rightarrow ex1p \rightarrow ex2p \rightarrow SCH5D))$$ □

X2 *SCH5D* poate da o combinaţie diferită de rest la fiecare utilizare. Iată în continuare un exemplu de automat care totdeauna dă aceeaşi combinaţie, dar nu ştim iniţial care va fi aceea (vezi 1.1.2 X3, X4).

$$SCH5E=SCH5A \sqcap SCH5B$$

Desigur, după ce automatul dă prima monedă comportarea ulterioară este complet previzibilă. Din acest motiv,

$$SCH5D \neq SCH5E$$ □

Nedeterminismul a fost introdus aici în forma sa cea mai pură şi simplă de operatorul binar ⊓. Desigur însă, ⊓ nu este prea util în *implementarea* unui proces. Ar fi o prostie să construim atât P cât şi Q, să-i punem într-o căciulă, să facem o alegere arbitrară între ei şi să aruncăm cealaltă alternativă. Principalul avantaj al nedeterminismului este în *specificarea* unui proces. Un proces specificat ca ($P \sqcap Q$) poate fi implementat atât prin construirea lui P cât şi a lui Q. Alegerea poate fi făcută în avans de cel care implementează pe motive nerelevante (şi deliberat ignorate) în specificare, ca de exemplu cost minim, timpi rapizi de răspuns. De fapt operatorul ⊓ nu va fi folosit prea des nici în specificaţii. Nedeterminismul provine mai natural din definirea altor operatori prezentaţi mai departe în acest capitol.

3.2.1 Legi

Legile algebrice care guvernează alegerea nedeterministă sunt foarte simple şi evidente. Alegerea între P şi P dă tot P.

L1 $P \sqcap P = P$ (idempotența)

Nu contează în ce ordine se prezintă alegerea.

L2 $P \sqcap Q = Q \sqcap P$ (simetria)

O alegere între trei procese poate fi împărțită în două alegeri binare. Nu contează în ce fel se face aceasta.

L3 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ (asociativitate)

Momentul în care se face alegerea nedeterministă nu este important. Un proces care întâi se angajează în x și apoi face o alegere nedeterministă este la fel cu unul care întâi face alegerea și apoi se angajează în x.

L4 $x \to (P \sqcap Q) = (x \to P) \sqcap (x \to Q)$ (distributivitate)

Legea L4 afirmă că operatorul prefix se distribuie în nedeterminism. Astfel de operatori se spune că sunt *distributivi*. Un operator diadic se spune că este distributiv dacă se distribuie cu $\sqcap$ independent în ambele argumente. Cei mai mulți din operatorii definiți până acum pentru procese sunt distributivi în acest sens.

L5 $(x{:}B \to (P(x) \sqcap Q(x))) = (x{:}B \to P(x)) \sqcap (x{:}B \to Q(x))$
L6 $P \| (Q \sqcap R) = (P \| Q) \sqcap (P \| R)$
L7 $(P \sqcap Q) \| R = (P \| R) \sqcap (Q \| R)$
L8 $f(P \sqcap Q) = f(P) \sqcap f(Q)$

Totuși operatorul de recursivitate *nu* este distributiv cu excepția cazului trivial când operanzii lui $\sqcap$ sunt identici. Aceasta se arată simplu prin diferența între cele două procese.

$$P = \mu X.((a \to X) \sqcap (b \to X))$$
$$Q = (\mu X.(a \to X)) \sqcap (\mu X.(b \to X))$$

P poate face o alegere independentă între a și b la fiecare iterație astfel că urmele sale includ

$$<a,b,b,a,b>$$

Q trebuie să facă o alegere între a se angaja în a sau b tot timpul astfel că urmele sale nu includ pe cea scrisă mai sus. Totuși P poate alege să se angajeze în a sau b tot timpul astfel că

$urme(Q) \leq urme(P)$

În unele teorii, nedeterminismul este obligat să fie *echitabil* în sensul că un eveniment care *poate* apărea infinit de des *trebuie* să apară (cu toate că nu există nici o limită a cât de mult poate fi întârziat). În teoria noastră nu există un astfel de concept de echitate. Deoarece observăm numai secvenţe finite ale comportării unui proces, dacă un eveniment poate fi amânat la infinit nu putem spune dacă va avea loc sau nu. Dacă vrem să insistăm ca evenimentul să apară trebuie să existe un număr n astfel că orice urmă mai mare ca n va conţine acel eveniment. Procesul în care se întâmplă acest lucru trebuie proiectat explicit pentru a satisface această constrângere. De exemplu, în procesul P_0 definit mai jos impunem ca evenimentul a să apară în mai puţin de n paşi de la apariţia anterioara.

$$\begin{cases} P_i = (a \to P_0) \sqcap (b \to P_{i+1}) & \text{pentru } 0 \leq i < n \\ P_n = (a \to P_0) \end{cases}$$

Mai târziu vom vedea că atât Q cât şi P_0 sunt implementări valide ale lui P.

Dacă se cere echitatea nedeterminismului aceasta poate fi specificată şi implementată la un nivel separat, de exemplu, prin atribuirea probabilităţilor diferite de zero alternativelor unei alegeri nedeterministe. Pare rezonabil de a separa raţionamentele probabilistice complexe de preocupările privind corectitudinea logică a comportării proceselor.

Datorită legilor L1–L3 este util de introdus un operator de alegere multiplă. Fie S o mulţime finită nevidă.

$$S = \{i, j, \ldots, k\}$$

Atunci definim

$$\prod_{x:S} P(x) = P(i) \sqcap P(j) \sqcap \ldots \sqcap P(k)$$

$\prod_{x:S}$ este fără sens când S este vidă sau infinită.

3.2.2 Implementări

Aşa cum am arătat mai sus unul din principalele motive pentru care am introdus nedeterminismul este diminuarea detaliilor implementărilor. Aceasta înseamnă că pot fi mai multe implementări ale unui proces nedeterminist P fiecare cu un mod observabil şi diferit de comportare. Diferenţele provin din

diferitele decizii nedeterministe inerente lui *P*. O anumită alegere poate fi făcută de cel care implementează înainte ca procesul să înceapă sau poate fi amânată până la derularea procesului.

De exemplu, o implementare a lui $(P \sqcap Q)$ ar fi selectarea primului operand

$$sau1(P,Q)=P$$

Altă implementare este obţinută prin selectarea celui de-al doilea operand pe motive de eficienţă mai mare pe o anumită maşină.

$$sau2(P,Q)=Q$$

În fine, o a treia implementare amână decizia până ce procesul începe să se execute. Apoi se permite mediului să facă alegerea, selectând un eveniment care este posibil pentru un proces dar nu şi pentru celălalt. Dacă evenimentul este posibil pentru ambele procese decizia este iarăşi amânată.

*sau*3(*P*,*Q*)=λ*x*.**if** *P*(*x*)="*BLIP* **then** *Q*(*x*)
else if *Q*(*x*)="*BLIP* **then** *P*(*x*)
else *sau*3(*P*(*x*),*Q*(*x*))

Aici am dat trei posibile implementări ale aceluiaşi operator. De fapt sunt mult mai multe : de exemplu o implementare care se comportă ca *sau*3 pentru primii 5 paşi, iar dacă toţi aceşti paşi sunt posibili atât pentru *P* cât şi pentru *Q* atunci se alege arbitrar *P*.

Deoarece proiectantul procesului $(P \sqcap Q)$ nu are controlul dacă *P* sau *Q* vor fi selectaţi, el trebuie să se asigure că acest sistem va funcţiona corect în ambele cazuri. Dacă există vreun risc ca *atât P cât şi Q* să se blocheze interacţionând cu mediul atunci $(P \sqcap Q)$ are de asemenea acest risc. Implementarea *sau*3 minimizează riscul de blocaj întârziind alegerea până ce o va face mediul şi apoi selectând dintre *P* sau *Q* pe cel care *nu* se blochează. Din acest motiv definiţia lui *sau*3 este cunoscută ca *nedeterminism angelic*. Dar preţul plătit este mare în raport cu eficienţa : dacă alegerea între *P* şi *Q* nu este făcută în primul pas, atât *P* cât şi *Q* trebuie să se execute concurent până ce mediul alege un eveniment care este posibil pentru unul dar nu şi pentru celălalt. În cazul simplu dar extrem *sau*3(*P*,*P*), aceasta nu se va produce şi ineficienţa va fi evident maximă.

În contrast cu *sau*3, implementările *sau*1 şi *sau*2 sunt asimetrice în sensul că

$$sau1(P,Q) \neq sau1(Q,P)$$

Aceasta pare să afecteze legea 3.2.1 L2, dar nu este aşa. Legile se aplică proceselor, nu unei implementări particulară a lor. De fapt ele afirmă identitatea *mulţimii tuturor* implementărilor membrilor lor stângi şi drepţi. De exemplu, deoarce *sau*3 este simetric

$$\{sau1(P,Q), sau2(P,Q), sau3(P,Q)\}=\{P,Q, sau3(P,Q)\}$$
$$=\{sau2(Q,P), sau1(Q,P), sau3(Q,P)\}$$

Unul din avantajele introducerii nedeterminismului este evitarea pierderii de simetrie care ar rezulta din selectarea uneia din cele două implementări simple şi totodată evitarea ineficienţei unei implementări simetrice gen *sau*3.

3.2.3 Urme

Dacă s este o urmă a lui P atunci s este o urmă posibilă pentru $(P\sqcap Q)$ în cazul că este selectat P. Similar dacă s este o urmă a lui Q, este de asemenea o urmă a lui $(P\sqcap Q)$. Reciproc, fiecare urmă a lui $(P\sqcap Q)$ trebuie să fie o urmă a uneia din cele două alternative. Comportarea lui $(P\sqcap Q)$ după s este definită de oricare din P sau Q s-ar putea angaja în s. Dacă ambele ar putea, alegerea rămâne nedeterministă.

L1 $urme(P\sqcap Q)=urme(P)\cup urme(Q)$

L2 $(P\sqcap Q)/s=Q/s$ dacă $s\in(urme(Q)-urme(P))$
$=P/s$ dacă $s\in(urme(P)-urme(Q))$
$=(P/s)\sqcap(Q/s)$ dacă $s\in(urme(P)\cap urme(Q))$

3.3 Alegere generală

Mediul în care evoluează $(P\sqcap Q)$ nu are controlul sau chiar cunoştinţa de alegerea care este făcută între P şi Q sau a momentului în care se face alegerea. Astfel $(P\sqcap Q)$ nu este o metodă bună de a combina procese din cauză că mediul trebuie să fie pregătit să trateze atât cu P cât şi cu Q, cu unul dintre ele separat fiind mult mai uşor de tratat. De aceea introducem o altă operaţie, $(P\square Q)$, pentru ca mediul să *poată* controla care dintre P şi Q va fi selectat, asigurându-se exercitarea controlului cu prima ocazie. Dacă evenimentul *nu* este un prim eveniment posibil pentru P atunci va fi selectat Q, iar dacă Q nu se poate angaja iniţial în eveniment va fi selectat P. Dacă totuşi primul eveniment este posibil atât pentru P cât şi pentru Q atunci alegerea între cele două procese este nedeterministă. (Desigur, dacă evenimentul este imposibil atât pentru P cât şi pentru Q, el nu poate să apară). Ca de obicei

$\alpha(P\Box Q)=\alpha P=\alpha Q$

În cazul când nici un eveniment inițial pentru P nu este posibil pentru Q, operatorul de alegere generală este același cu | care a fost folosit pentru a reprezenta alegerea între două evenimente diferite

$(c\to P\Box d\to Q)=(c\to P|d\to Q)$ dacă $c\neq d$

Dacă totuși evenimentele inițiale sunt aceleași, $(P\Box Q)$ degenerează în alegere nedeterministă

$(c\to P\Box c\to Q)=(c\to P)\sqcap(c\to Q)$

Aici am adoptat prioritatea lui $\to$ față de $\Box$.

3.3.1 Legi

Legile algebrice pentru $\Box$ sunt similare cu acelea pentru $\sqcap$ iar motivele sunt aceleași

L1 - L3 $\Box$ este idempotent, simetric și asociativ

L4 $P\Box STOP=P$

Următoarea lege formalizează definiția explicativă a operatorului

L5 $(x{:}A\to P(x))\Box(y{:}B\to Q(y))=(z{:}(A\cup B)\to($**if** $z\in(A-B)$ **then** $P(z)$
else if $z\in(B-A)$ **then** $Q(z)$
else if $z\in(A\cap B)$ **then** $(P(z)\sqcap Q(z))))$

Ca toți ceilalți operatori introduși până acum (în afară de recursivitate), $\Box$ distribuie față de $\sqcap$

L6 $P\Box(Q\sqcap R)=(P\Box Q)\sqcap(P\Box R)$

Mai surprinzător este că $\sqcap$ distribuie față de $\Box$

L7 $P\sqcap(Q\Box R)=(P\sqcap Q)\Box(P\sqcap R)$

Această lege afirmă că alegerile nedeterministe și cele făcute de mediu sunt independente în sensul că selecția făcută într-un fel nu este influențată de cealaltă parte. Fie John agentul care face alegerile nedeterministe și fie Mary me-

diul. În partea stângă a legii, John alege (⊓) între P și a o lăsa pe Mary să aleagă între Q și R. În partea dreaptă Mary are posibilitatea

atât (1) să-i ofere lui John alegerea între P și Q
cît și (2) să-i ofere lui John alegerea între P și R.

În ambele părți ale ecuației, dacă John alege P, atunci P va fi alegerea generală. Dacă John nu-l selectează pe P alegerea între Q și R este făcută de Mary. Astfel rezultatele alegerilor descrise în partea stângă și dreaptă a legii sunt totdeauna egale. Desigur, același raționament se aplică la L6.

Explicația dată mai sus este într-un fel subtilă. Desigur se va înțelege mai bine legea ca o consecință inevitabilă a altor legi și definiții mai evidente pe care le vom da în continuarea acestui capitol.

3.3.2 Implementare

Implementarea operatorului alegere generală rezultă imediat din legea L5. Presupunând simetria lui *sau*, este de asemenea simetrică funcția

alternat(P,Q)= λ*x*. **if** $P(x)$="*BLIP* **then** $Q(x)$
else if $Q(x)$="*BLIP* **then** $P(x)$
else *sau*($P(x)$,$Q(x)$)

3.3.3 Urme

Fiecare urmă a lui (P□Q) trebuie să fie o urmă a lui P sau a lui Q și reciproc

L1 *urme*(P□Q)=*urme*(P)∪*urme*(Q)

Următoarea lege este puțin diferită față de legea corespunzătoare pentru ⊓

L2 (P□Q)/s=P/s dacă $s \in$ *urme*(P)−*urme*(Q)
=Q/s dacă $s \in$ *urme*(Q)−*urme*(P)
=(P/s)⊓(Q/s) dacă $s \neq$ <> și $s \in$ *urme*(P)∩*urme*(Q)

3.4 Refuzuri

Distincția între (P⊓Q) și (P□Q) este subtilă. Ele nu se pot distinge prin urmele lor din cauză că fiecare urmă a unuia este o posibilă urmă a celuilalt. Totuși este posibil să le punem într-un mediu în care (P⊓Q) se poate bloca la primul pas, în timp ce (P□Q) nu. În următorul exemplu fie $x \neq y$ și

$P=(x \rightarrow P)$, $Q=(y \rightarrow Q)$, $\alpha P=\alpha Q=\{x,y\}$

Apoi $(P \Box Q) \| P=(x \rightarrow P)$
$=P$
dar $(P \sqcap Q) \| P=(P \| P) \sqcap (Q \| P)$
$=P \sqcap STOP$

Aceasta ne arată că în mediul P, $(P \sqcap Q)$ poate ajunge la blocaj dar $(P \Box Q)$ nu. Desigur, chiar cu $(P \sqcap Q)$ nu putem fi siguri că nu va apare blocaj. Dacă nu apare, nu vom şti niciodată dacă ar fi putut apărea. Numai simpla posibilitate a apariţiei blocajului şi este suficientă pentru a distinge $(P \Box Q)$ de $(P \sqcap Q)$.

În general, fie X o mulţime de evenimente oferite iniţial de mediu unui proces P, care în acest context are acelaşi alfabet cu alfabetul lui P. Dacă este posibil pentru P să se blocheze în primul pas când este plasat în acest mediu, spunem că X este un *refuz* al lui P. Mulţimea tuturor acestor refuzuri ale lui P este notată

$refuzuri(P)$

De remarcat că refuzurile unui proces constituie o familie de mulţimi de simboluri. Aceasta este o complexitate nefericită, dar pare să fie de neînlăturat într-o manieră adecvată de tratare a nedeterminismului. În loc de refuzuri ar fi mai natural să folosim mulţimea de simboluri pe care un proces este *gata* să le accepte, totuşi refuzurile sunt mai simple din cauză că se supun legilor L9 şi L10 din paragraful 3.4.1 (mai jos) în timp ce legile corespunzătoare pentru mulţimile *gata* ar fi mai complicate.

Introducerea conceptului de refuz permite să fie făcută o distincţie clară şi formală între procesele deterministe şi nedeterministe. Un proces se spune că este *determinist* dacă nu refuză nici un eveniment în care se poate angaja. Cu alte cuvinte, o mulţime este un refuz pentru un proces determinist numai dacă mulţimea nu conţine nici un eveniment în care procesul să se poată angaja iniţial, sau mai formal

P este determinist $\Rightarrow (X \in refuzuri(P) \equiv (X \cap P^0=\{\}))$
unde $P^0=\{x \mid \langle x \rangle \in urme(P)\}$

Această condiţie se aplică nu numai la pasul iniţial al lui P dar de asemenea după orice secvenţă posibilă de evenimente pentru P. Astfel putem defini

P este determinist $\equiv \forall s{:}urme(P).(X \in refuzuri(P/s) \equiv (X \cap (P/s)^0=\{\}))$

Un proces nedeterminist nu suferă de această proprietate, astfel că dacă există în orice moment cel puţin un eveniment în care să se poată angaja, de asemenea (ca rezultat al unei alegeri interne nedeterministe) poate refuza angajarea în acest eveniment chiar dacă mediul său cu care interacţionează este pregătit să se angajeze.

3.4.1 Legi

Următoarele legi definesc refuzurile pentru diverse procese simple. Procesul *STOP* nu face nimic şi refuză totul.

L1 $refuzuri(STOP_A)$=toate submulţimile lui A (inclusiv A)

Un proces $(c \rightarrow P)$ refuză orice mulţime care nu conţine evenimentul c

L2 $refuzuri(c \rightarrow P)=\{X \mid X \subseteq (\alpha P - \{c\})\}$

Aceste două legi se generalizează în

L3 $refuzuri(x{:}B \rightarrow P(x))=\{X \mid X \subseteq (\alpha P - B)\}$

Dacă P poate refuza X la fel va face şi $(P \sqcap Q)$ dacă este selectat P. Similar orice refuz al lui Q este de asemenea un posibil refuz al lui $(P \sqcap Q)$. Acestea sunt singurele refuzuri astfel că

L4 $refuzuri(P \sqcap Q)=refuzuri(P) \cup refuzuri(Q)$

O argumentaţie inversă se aplică la $(P \square Q)$. Dacă X *nu* este un refuz al lui P, atunci P *nu poate* refuza pe X şi deci nici $(P \square Q)$. Similar dacă X nu este un refuz al lui Q, nu este un refuz nici al lui $(P \square Q)$. Totuşi, dacă *atât P cât şi Q* pot refuza pe X, atunci şi $(P \square Q)$ poate.

L5 $refuzuri(P \square Q)=refuzuri(P) \cap refuzuri(Q)$

Compararea lui L5 cu L4 ne dă diferenţa între $\square$ şi $\sqcap$.

Dacă P poate refuza X şi Q poate refuza Y atunci combinaţia lor $(P \| Q)$ poate refuza toate evenimentele refuzate de P precum şi toate evenimentele refuzate de Q, cu alte cuvinte poate refuza reuniunea mulţimilor X şi Y.

L6 $refuzuri(P \| Q)=\{X \cup Y \mid X \in refuzuri(P) \wedge Y \in refuzuri(Q)\}$

Pentru schimbarea de simbol legea este clară

L7 $refuzuri(f(P))=\{f(X) \mid X \in refuzuri(P)\}$

Există un număr de legi generale privind refuzurile. Un proces poate refuza numai evenimentele din propriul său alfabet. Un proces se blochează când mediul nu-i oferă evenimente. Dacă un proces refuză o mulţime nevidă el poate refuza orice submulţime a acelei mulţimi. În fine, orice eveniment *x* care nu apare ca refuz iniţial poate fi adăugat la orice mulţime *X* deja refuzată.

L8 $X \in refuzuri(P) \Rightarrow X \subseteq \alpha P$
L9 $\{\} \in refuzuri(P)$
L10 $(X \cup Y) \in refuzuri(P) \Rightarrow X \in refuzuri(P)$
L11 $X \in refuzuri(P) \Rightarrow (X \cup \{x\}) \in refuzuri(P) \vee \langle x \rangle \in urme(P)$ dacă $x \in \alpha P$

3.5 Mascare (ascundere)

În general, alfabetul unui proces conţine numai acele evenimente care se consideră a fi relevante iar apariţiile acestor evenimente necesită participarea simultană a mediului. În descrierea comportării interne a unui sistem adeseori avem nevoie să considerăm evenimente reprezentând tranziţii interne acelui sistem. Astfel de evenimente pot exprima interacţiunile şi comunicaţiile între componentele active concurente din care este construit sistemul, de exemplu *LANŢ2* (2.6.X4) şi 2.6.2.X3. După construirea sistemului, mascăm structura componentelor. De asemenea dorim să mascăm toate apariţiile acţiunilor interne sistemului. De fapt vrem ca aceste acţiuni să apară automat şi instantaneu cât mai curând posibil fără a fi observate sau controlate de mediul procesului. Dacă *C* este o mulţime finită de evenimente gata de a fi mascate în acest mod, atunci

$P \backslash C$

este un proces care se comportă ca *P* cu excepţia faptului că orice apariţie a unui eveniment din *C* este mascată. De fapt intenţia este ca

$\alpha(P \backslash C)=(\alpha P)-C$

Exemple

X1 Un automat de vândut zgomotos (2.3.X1) poate fi plasat într-o carcasă antifonică

$AVSUNET \backslash \{bing, bang\}$

Poate fi înlăturată din alfabet şi posibilitatea de a livra bomboane fără a afecta comportarea actuală. Procesul care rezultă este automatul simplu

$$AVS = AVSUNET\setminus\{bing, bang, bonbon\}$$ □

Când două procese au fost combinate pentru a funcţiona concurent, interacţiunile lor comune sunt de obicei privite ca activităţi interne sistemului rezultat. Ele se preconizează să apară autonom şi cât mai repede posibil fără cunoştinţa sau intervenţia mediului exterior sistemului. Astfel, simbolurile din intersecţia celor două alfabete ale componentelor trebuie să fie mascate.

X2 Fie $\alpha P=\{a,c\}$, $\alpha Q=\{b,c\}$, $P=(a\to c\to P)$, $Q=(c\to b\to Q)$ (2.3.1.X1)

Acţiunea c în alfabetele lui P şi Q este acum privită ca o acţiune internă ce trebuie mascată

$$(P\|Q)\setminus\{c\}=(a\to c\to \mu X.(a\to b\to c\to X\,|\,b\to a\to c\to X))\setminus\{c\}$$
$$=a\to \mu X.(a\to b\to X\,|\,b\to a\to X)$$ □

3.5.1 Legi

Primele legi stabilesc că mascarea nici unui simbol nu are efect şi că nu este nici o diferenţă în ce ordine sunt mascate simbolurile unei mulţimi. Celelalte legi din acest grup ne arată cum mascarea se distribuie cu ceilalţi operatori. Mascarea a nimic lasă procesul neschimbat.

L1 $P\setminus\{\}=P$

Pentru a masca o mulţime de simboluri şi ulterior altele este totuna cu a le masca simultan

L2 $(P\setminus B)\setminus C=P\setminus(B\cup C)$

Mascarea se distribuie obişnuit cu alegerea nedeterministă

L3 $(P\sqcap Q)\setminus C=(P\setminus C)\sqcap(Q\setminus C)$

Mascarea nu afectează un proces oprit ci numai alfabetul său

L4 $STOP_A\setminus C=STOP_{A-C}$

Scopul mascării este de a permite oricărui eveniment mascat să apară imediat şi independent, în condiţiile invizibilităţii acestor apariţii. Evenimentele nemascate rămân neschimbate.

L5 $(x \to P) \backslash C = x \to (P \backslash C)$ dacă $x \notin C$
$= P \backslash C$ dacă $x \in C$

Dacă C conţine numai evenimente în care P şi Q participă independent mascarea lui C se distribuie în compunerea lor concurentă

L6 Dacă $\alpha P \cap \alpha Q \cap C = \{\}$
atunci $(P \| Q) \backslash C = (P \backslash C) \| (Q \backslash C)$

Legea de mai sus nu este prea utilă în mod normal din cauză că ceea ce dorim să mascăm sunt interacţiunile dintre procesele concurente, cu alte cuvinte mulţimea $\alpha P \cap \alpha Q$ a evenimentelor comune în care participă.

Mascarea se distribuie evident cu schimbarea de simbol dată de o funcţie injectivă

L7 $f(P \backslash C) = f(P) \backslash f(C)$

Dacă nici unul din evenimentele posibile iniţial ale alegerii nu este mascat atunci alegerea iniţială rămâne aceeaşi ca şi înainte de mascare.

L8 Dacă $B \cap C = \{\}$
atunci $(x{:}B \to P(x)) \backslash C = (x{:}B \to (P(x) \backslash C))$

Ca şi operatorul alegere generală □, mascarea evenimentelor poate introduce nedeterminism. Când pot să apară mai multe evenimente mascate diferite nu este precizat care va apare, dar oricare va apare este mascat.

L9 Dacă $B \subseteq C$ şi B este finit şi nevid

atunci $(x{:}B \to P(x)) \backslash C = \prod_{x:B} (P(x) \backslash C)$

Într-un caz intermediar când o parte din evenimentele iniţiale sunt mascate şi altele nu, situaţia este ceva mai complicată. Considerăm procesul

$(c \to P | d \to Q) \backslash C$ unde $c \in C$, $d \notin C$

Evenimentul mascat c poate apărea imediat. În acest caz întreaga comportarea va fi definită de $(P \backslash C)$ şi posibilitatea apariţiei evenimentului d va fi eliminată.

Dar nu putem presupune sigur că d nu va apărea. Dacă mediul este pregătit să o facă, d poate foarte bine apărea înaintea evenimentului mascat, după care evenimentul mascat c poate să nu mai apară. Dar chiar dacă d apare, ar putea fi angajat de procesul $(P\backslash C)$ după apariţia mascată a lui c. În acest caz, comportarea ce rezultă este definită de

$$(P\backslash C)\square(d\rightarrow(Q\backslash C))$$

Alegerea între membrul drept şi $(P\backslash C)$ este nedeterministă. Desigur aceasta este o justificare cam sinuoasă pentru destul de complexa lege

$$(c\rightarrow P|d\rightarrow Q)\backslash c=(P\backslash C)\sqcap((P\backslash C)\square(d\rightarrow(Q\backslash c)))$$

Raţionamente similare justifică legea mai generală

L10 Dacă $C\cap B\neq\{\}$ şi este finită
atunci $(x:B\rightarrow P(x))\backslash C=Q\sqcap(Q\square(x:(B-C)\rightarrow(P(x)\backslash C))$

unde $Q=\prod_{x:B\cap C}(P(x)\backslash C)$

O reprezentare a acestor legi este dată în paragraful 3.5.4.

De notat că $\backslash C$ nu se distribuie retroactiv cu $\square$. Un contraexemplu este

$$
\begin{aligned}
(c\rightarrow STOP\square d\rightarrow STOP)\backslash\{c\} &= STOP\sqcap STOP\square(d\rightarrow STOP)) && \text{L10}\\
&= STOP\sqcap(d\rightarrow STOP) && \text{3.3.1 L4}\\
&\neq d\rightarrow STOP\\
&= STOP\square(d\rightarrow STOP)\\
&= ((c\rightarrow STOP)\backslash\{c\})\square((d\rightarrow STOP)\backslash\{c\})
\end{aligned}
$$

Mascarea reduce alfabetul unui proces. De asemenea putem defini o operaţie care extinde alfabetul unui proces prin includerea simbolurilor unei mulţimi B.

$$\alpha(P_{+B})=\alpha P\cup B$$
$$P_{+B}=(P\|STOP_B) \qquad \text{dacă } B\cap\alpha P=\{\}$$

Nici unul din noile evenimentele din B nu va apare de fapt, astfel că comportarea lui P_{+B} este efectiv aceeaşi cu aceea a lui P.

L11 $urme(P_{+B})=urme(P)$

Prin urmare mascarea lui B este inversul extinderii alfabetului prin B.

L12 $(P_{+B})\backslash B=P$

Este cazul să ne punem aici o problemă ce va fi rezolvată mai târziu în paragraful 3.8. În cazuri simple, mascarea se distribuie cu recursivitatea

$$\mu X{:}A.(c\rightarrow X))\backslash\{c\}=\mu X{:}(A-\{c\}).((c\rightarrow X_{+\{c\}})\backslash\{c\})$$
$$=\mu X{:}(A-\{c\}).X \qquad \text{din L12, L5}$$

Astfel încercarea de a masca o secvență *infinită* de evenimente consecutive ne conduce la același rezultat nefavorabil ca și o buclă infinită sau o recursivitate fără gardă. Termenul general pentru acest lucru este *divergență*.

Aceeași problemă apare chiar dacă procesul divergent este capabil de a se angaja la infinit într-un eveniment nemascat, de exemplu

$$\mu X.(c\rightarrow X\square d\rightarrow P))\backslash\{c\}=\mu X.((c\rightarrow X\square d\rightarrow P)\backslash\{c\})$$
$$=\mu X.(X\backslash\{c\})\sqcap((X\backslash\{c\})\square d\rightarrow(P\backslash\{c\})) \qquad \text{din L10}$$

Din nou, recursivitatea este fără gardă și ne conduce la divergență. Chiar și așa, apare că mediul oferă infinit posibilitatea selectării lui d și nu există nici o metodă de a preveni procesul în a alege la infinit angajarea evenimentului mascat. Această posibilitate pare să conducă la realizarea celei mai mari eficiențe în implementare. De asemenea chestiunea este într-un fel legată de decizia noastră de a nu insista pe echitatea nedeterminismului, așa cum s-a prezentat în paragraful 3.2.1. O discuție mai riguroasă despre divergență se va face în paragraful 3.8.

Există un sens, unul important, în care mascarea este de fapt echitabilă. Fie $d\in\alpha R$ și considerăm procesul

$$((c\rightarrow a\rightarrow P|d\rightarrow STOP)\backslash\{c\})\|(a\rightarrow R)$$
$$=((a\rightarrow P\backslash\{c\})\sqcap(a\rightarrow P\backslash\{c\}\square d\rightarrow STOP))\|(a\rightarrow R) \qquad \text{din L10}$$
$$=(a\rightarrow P\backslash\{c\})\|(a\rightarrow R)\sqcap(a\rightarrow P\backslash\{c\}\square d\rightarrow STOP)\|(a\rightarrow R)$$
$$=a\rightarrow((P\backslash\{c\})\|R)$$

Acest exemplu ne arată că un proces care oferă alegerea între o acțiune mascată c și una nemascată d, nu poate insista să apară acțiunea nemascată. Dacă mediul (în acest exemplu, $a\rightarrow R$) nu este pregătit pentru d, atunci evenimentul mascat trebuie să apară, astfel că mediul are șansa să interacționeze cu procesul care rezultă (în exemplu $(a\rightarrow P\backslash\{c\})$).

3.5.2 Implementare

Pentru simplitate vom implementa o operaţie care maschează un singur simbol odată

$$masc(P,c)=P\backslash\{c\}$$

O mulţime de simboluri pot fi mascate unul după altul

$$P\backslash\{c_1,c_2,\ldots,c_n\}=(\ldots\ldots((P\backslash\{c_1\}\backslash\{c_2\})\backslash \ \ldots\ )\backslash\{c_n\}$$

Cea mai simplă implementare este aceea care face totdeauna ca evenimentul mascat să apară invizibil ori de câte ori poate şi cât mai curând posibil.

masc(*P*,*c*)= **if** *P*(*c*)="*BLIP* **then**
(λ*x*. **if** *P*(*x*)="*BLIP* **then** "*BLIP*
else *masc*(*P*(*x*),*c*))
else *masc*(*P*(*c*),*c*)

Să vedem ce se întâmplă când funcţia de mascare este aplicată unui proces capabil de a se angaja într-o secvenţă infinită de evenimente mascate, de exemplu

$$masc(\mu X.(c\rightarrow X\Box d\rightarrow P),c)$$

În acest caz, testul (*P*(*c*)="*BLIP*) va genera totdeauna *FALSE*, astfel că funcţia *masc* va selecta mereu ramura else, apelându-se de aceea imediat recursiv. Nu există ieşire din această recursivitate, astfel că nu va apare comunicare cu lumea exterioară. Acesta este preţul pentru încercarea de a implementa un proces divergent.

Prezenta implementare a mascării nu se supune la L2. Într-adevăr, ordinea în care se maschează simbolurile este importantă, aşa cum se arată în exemplul

$$P=(c\rightarrow STOP|d\rightarrow a\rightarrow STOP)$$

Atunci $masc(masc(P,c),d)=masc(masc(STOP,c),d)$
$=STOP$

şi $masc(masc(P,d),c)=masc(masc((a\rightarrow STOP),d),c)$
$=(a\rightarrow STOP)$

Dar aşa cum s-a explicat în paragraful 3.2.2, o implementare particulară a unui operator nedeterminist nu trebuie să se supună legilor. Este suficient ca ambele rezultate arătate mai sus să permită implementări ale aceluiaşi proces

$$P\backslash\{c,d\}=(STOP\sqcap(a\rightarrow STOP))$$

3.5.3 Urme

Dacă t este o urmă a lui P, urma corespunzătoare lui $P\backslash C$ este obţinută simplu din t prin eliminarea tuturor apariţiilor simbolurilor din C. Reciproc, fiecare urmă a lui $P\backslash C$ trebuie să se obţină dintr-o astfel de urmă a lui P. De aceea formulăm

L1 $urme(P\backslash C)=\{t\restriction(\alpha P-C) \mid t\in urme(P)\}$ când $\forall s{:}urme(P).\neg diverge(P/s,C)$

Condiţia $diverge(P,C)$ semnifică divergenţa imediată a lui P odată cu mascarea lui C, cu alte cuvinte P se poate angaja într-o secvenţă nemărginită de evenimente mascate. De aceea definim

$$diverge(P,C)=\forall n.\exists s{:}urme(P)\cap C^{*}.\#s>n$$

Corespunzător unei singure urme s a lui $P\backslash C$ pot fi mai multe urme t ale comportării posibile în care se angajează P şi care nu pot fi diferenţiate după mascare, adică $t\restriction(\alpha P-C)=s$. Următoarea lege formulează că după parcurgerea urmei s nu se poate şti care din comportările ulterioare ale lui P vor defini comportarea ulterioară a lui $(P\backslash C)$.

L2 $(P-C)/s=(\prod_{t:T} P/t)\backslash C$

unde $T=urme(P)\cap\{t \mid t\restriction(\alpha P-C)=s\}$
în condiţiile când T este finit şi $s\in urme(P\backslash C)$

Aceste legi sunt restricţionate la cazul când procesul nu diverge. Restricţiile nu sunt importante din cauză că divergenţa nu se intenţionează a fi niciodată rezultatul definirii unui proces. Pentru o completă tratare a divergenţei a se vedea paragraful 3.8.

3.5.4. Reprezentări

Alegerea nedeterministă poate fi reprezentată printr-un nod din care ies două sau mai multe arce neetichetate. Ajungând în acel nod un proces trece imper-

ceptibil de-a lungul unuia din arce, alegerea fiind nedeterministă. Astfel $P \sqcap Q$ este reprezentat în fig. 3.1.

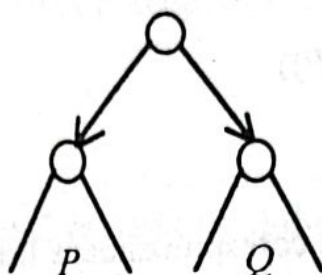


Figura 3.1

Legile algebrice ce guvernează nedeterminismul presupun identități între aceste reprezentări, de exemplu asociativitatea lui $\sqcap$ este ilustrată în fig. 3.2.

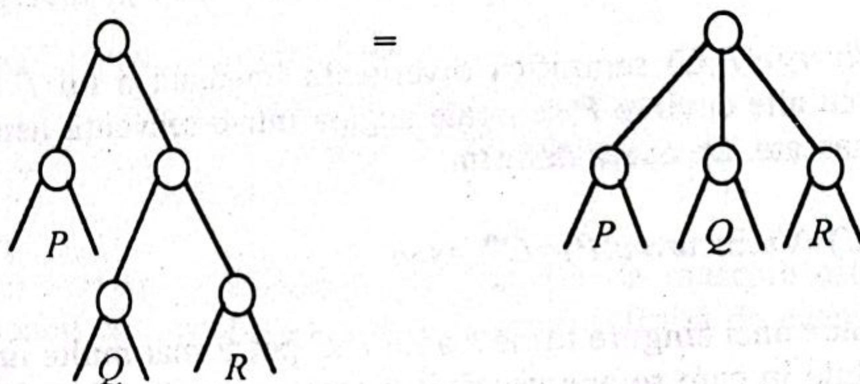


Figura 3.2

Mascarea simbolurilor poate fi privită ca o operație care pur și simplu elimină simbolurile respective din toate arcele pe care le etichetează astfel că aceste arce se transformă în arce neetichetate. Nedeterminismul rezultant decurge natural ca în fig. 3.3.

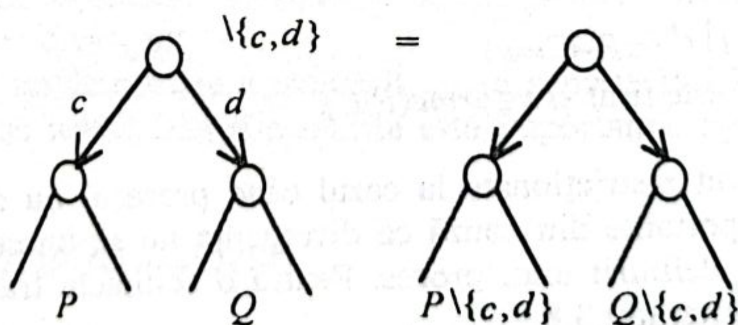


Figura 3.3

Dar care este semnificația unui nod când o parte din arcele sale sunt etichetate și altele nu? Răspunsul este dat de legea 3.5.1 L10. Un astfel de nod poate fi eliminat prin redesenare așa cum se arată în fig. 3.4.

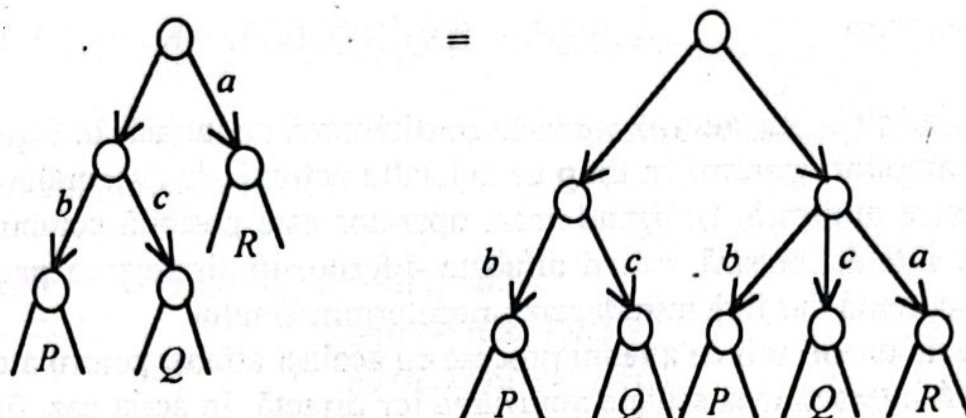


Figura 3,4

Este în general evident că astfel de eliminări sunt totdeauna posibile pentru arbori finiţi. Ele sunt posibile şi pentru grafuri infinite în condiţiile când graful nu conţine drumuri infinite din arce consecutive neetichetate ca de exemplu în fig. 3.5. O astfel de reprezentare apare numai în cazul divergenţei pe care am decis deja să o privim ca o eroare.

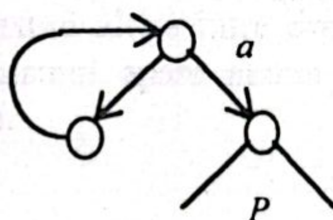


Figura 3.5

Ca un rezultat al aplicării transformării L10 este posibil ca un nod să poată dobândi două arce cu aceeaşi etichetă. Astfel de noduri pot fi eliminate cu legea dată la sfârşitul paragrafului 3.3 (fig. 3.6)

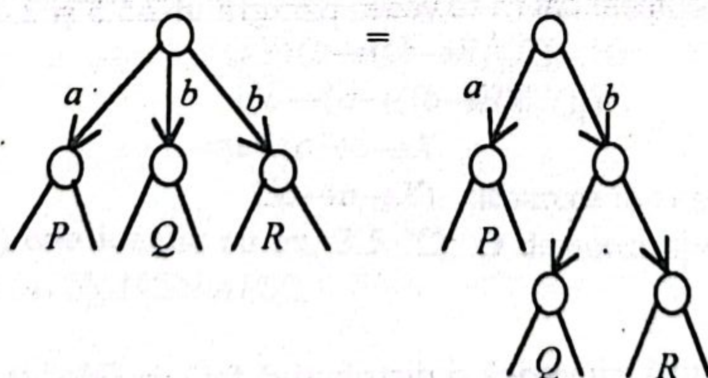


Figura 3.6

Reprezentările proceselor şi legile care le guvernează sunt incluse aici ca un ajutor în memorizarea şi înţelegerea lor. Aceste tehnici grafice nu se intenţionează a fi utilizate în transformările sau manipulările cu procese extinse.

3.6 Întreţeserea

Operatorul || definit în capitolul 2 necesită participarea simultană în acţiunile din alfabetul ambilor operanzi în timp ce celelalte acţiuni ale sistemului apar într-o întreţesere arbitrară. Utilizând acest operator este posibilă combinarea proceselor ce interacţionează, având alfabete diferite, în sisteme ce prezintă activitate concurentă dar fără introducerea nedeterminismului.

Totuşi este uneori util de a reuni procese cu acelaşi alfabet pentru a opera concurent fără interacţiunea sau sincronizarea lor directă. În acest caz, fiecare acţiune a sistemului este o acţiune exact a unui proces. Dacă unul dintre procese nu se poate angaja în acţiune atunci trebuie să o facă celălalt, dar dacă ambele procese s-ar putea angaja în acţiune, alegerea este nedeterministă. Această formă de compunere a proceselor este notată

$P|||Q$ (*P* întreţesut cu *Q*)

şi alfabetul său este definit normal

$\alpha(P|||Q)=\alpha P=\alpha Q$

Exemple

X1 Un *AVS* va accepta până la două monede înainte de a da până la două ciocolate (1.1.3 X6)

$(AVS|||AVS)=AVS2$ □

X2 Un valet format din patru valeţi tineri subalterni, fiecare servind numai un singur filozof la un moment dat (a se vedea paragraful 2.5.3 şi 2.6.4 X1)

$E|||E|||E|||E$

unde $E=VAL_T_M$ □

3.6.1 Legi

L1 - 3 ||| este asociativă, simetrică şi distributivă faţă de ⊓

L4 $P|||STOP=P$

L5 $P|||RUN=RUN$ în condiţiile în care *P* nu diverge

L6 $(x\to P)|||(y\to Q)=(x\to(P|||(y\to Q))\square y\to((x\to P)|||Q))$

L7 Dacă $P=(x:A\to P(x))$

şi $Q=(y:B\to Q(y))$

atunci $P|||Q = (x{:}A \rightarrow (P(x)|||Q) \Box (y{:}B \rightarrow (P|||Q(y)))$

De remarcat că ||| nu se distribuie cu □. Aceasta este arătat prin următorul contraexemplu (unde $b \neq c$)

$$((a \rightarrow STOP)|||(b \rightarrow Q \Box c \rightarrow R))/\langle a\rangle =$$
$$= (b \rightarrow Q \Box c \rightarrow R)$$
$$\neq ((b \rightarrow Q) \sqcap (c \rightarrow R))$$
$$= ((a \rightarrow STOP|||b \rightarrow Q) \Box (a \rightarrow STOP|||c \rightarrow R))/\langle a\rangle$$

În prima expresie din lanţul de egalităţi, apariţia lui a poate să implice progresul numai operandului din stânga lui |||, astfel că nu există nedeterminism. Operandul din stânga se opreşte şi alegerea între b şi c este lăsată la latitudinea mediului. În ultima expresie a lanţului de egalităţi, evenimentul a poate fi al oricărui operand al lui |||, de aceea alegerea este nedeterministă. Astfel mediul nu poate alege dacă următorul eveniment este b sau c.

L6 şi L7 afirmă că mediul alege între evenimentele iniţiale oferite de operanzii lui |||. Nedeterminismul apare numai când evenimentul ales este posibil pentru ambii operanzi.

Exemplu

X1 Fie $R = (a \rightarrow b \rightarrow R)$

Atunci $(R|||R) = (a \rightarrow ((b \rightarrow R)|||R) \Box a \rightarrow (R|||(b \rightarrow R)))$ L6

$= a \rightarrow ((b \rightarrow R)|||R) \sqcap (R|||(b \rightarrow R))$

$= a \rightarrow ((b \rightarrow R)|||R)$ L2

De asemenea $((b \rightarrow R)|||R) = a \rightarrow ((b \rightarrow R)|||(b \rightarrow R)) \Box b \rightarrow (R|||R)$ L6

$= (a \rightarrow (b \rightarrow ((b \rightarrow R)|||R))$ ca mai

$\Box b \rightarrow (a \rightarrow ((b \rightarrow R)|||R)))$ sus

$= \mu X.(a \rightarrow b \rightarrow X$

$\Box b \rightarrow a \rightarrow X)$ deoarece recursivitatea este cu gardă.

Astfel $(R|||R)$ este identic cu ex. 3.5 X2. O demonstraţie similară ne duce la concluzia că $(AVS|||AVS) = AVS2$. □

3.6.2. Urme şi refuzuri

O urmă a lui $(P|||Q)$ este o înteţesere arbitrară a unei urme din P cu o urmă din Q. Pentru o definiţie a întreţeserii a se vedea paragraful 1.9.3.

L1 $urme(P|||Q) = \{s \mid \exists t \in urme(P). \exists u \in urme(Q).\ s \text{ întreţese } (t,u)\}$

$(P|||Q)$ se poate angaja în orice acțiune inițială posibilă atât pentru P cât și pentru Q și de aceea poate refuza numai acele mulțimi care sunt refuzate atât de P cât și de Q

L2 $refuzuri(P|||Q)=refuzuri(P\Box Q)$

Comportarea lui $(P|||Q)$ după angajarea în evenimentele urmei s este definită de formula cam elaborată

L3 $(P|||Q)/s=\prod_{(t,u)\in T}(P/t)|||(Q/u)$

unde $T=\{(t,u)\mid t\in urme(P)\wedge u\in urme(Q)\wedge s \text{ întrețese } (t,u)\}$

Această lege reflectă faptul că nu există metodă de a afla în ce fel o urmă s a lui $(P|||Q)$ a fost construită ca o întrețesere a unei urme a lui P și a uneia a lui Q. Astfel că după s, comportarea ulterioară a lui $(P|||Q)$ poate reflecta orice întrețesere posibilă. Alegerea momentelor când se face întrețeserea nu este știută și nici determinată.

3.7 Specificații

În paragraful 3.4 am văzut necesitatea de a introduce mulțimile de refuzuri ca unul din aspectele observabile indirect ale comportării unui proces. În specificarea unui proces avem nevoie prin urmare de descrierea proprietăților dorite ale mulțimilor sale de refuzuri, la fel ca și pentru urme. Să utilizăm variabila *ref* pentru o mulțime de refuz arbitrară a unui proces la fel cum am folosit *ur* pentru a nota urma unui proces. Ca un rezultat, când P este un proces nedeterminist înțelesul său

P **sat** $S(ur,ref)$

este $\forall ur,ref\,.\,ur\in urme(P)\wedge ref\in refuzuri(P/ur)\Rightarrow S(ur,ref)$

Exemple

X1 Când un AV a primit mai multe monezi decât ciocolate poate furniza, utilizatorul specifică că nu trebuie să refuze eliberarea unei ciocolate

$ECHIT=(ur\downarrow choc<ur\downarrow mon\Rightarrow choc\notin ref)$

Este de la sine înțeles că *orice* urmă *ur* și *orice* refuz *ref* al unui proces specificat, totdeauna va satisface această specificație. □

X2 Când un *AV* a dat tot atâtea ciocolate câți bani s-au plătit, proprietarul specifică că nu trebuie să se refuze următoarea monedă

$PROFIT1=(ur\downarrow choc=ur\downarrow mon\Rightarrow mon\notin ref)$ □

X3 Un *AVS* ar trebui să satisfacă ambele specificații

$SPECNOI=ECHIT\wedge PROFIT1\wedge(ur\downarrow choc\leq ur\downarrow mon)$

Această specificație este satisfăcută de *AVS*. Este de asemenea satisfăcută de *AVS*2 (1.1.3 X6) care va accepta câteva monezi la rând și apoi va da câteva ciocolate. □

X4 Dacă se dorește, se poate limita numărul de monezi ce pot fi acceptate una după alta

$CELMULT2=(ur\downarrow mon-ur\downarrow choc\leq 2)$ □

X5 Dacă se dorește, automatul poate accepta cel puțin două monezi odată ori de câte ori cumpărătorul oferă

$CELPUȚIN2=(ur\downarrow mon-ur\downarrow choc<2\Rightarrow mon\notin ref)$ □

X6 Procesul *STOP* refuză orice eveniment din alfabetul său. Următorul predicat specifică cum un proces cu alfabetul *A* nu se va opri niciodată

$NONSTOP=(ref\neq A)$

Dacă *P* **sat** *NONSTOP* și dacă mediul permite angajarea în toate evenimentele din *A*, procesul *P* trebuie să se angajeze în unul din ele. Deoarece (vezi X3 de mai înainte)

$SPECNOIAV\Rightarrow ref\neq\{mon,choc\}$

rezultă că orice proces care satisface *SPECNOIAV* nu se va opri. □

Aceste exemple ne arată cum introducerea lui *ref* în specificarea unui proces permite exprimarea unui număr de proprietăți subtile și importante. Poate cea mai importantă dintre ele este aceea că un proces nu trebuie să se

oprească (X6). Aceste avantaje sunt obţinute cu preţul unei complexităţi mărite în demonstraţii şi reguli.

De asemenea se doreşte să se dovedească că un proces nu diverge. Paragraful 3.8 descrie cum un proces divergent poate face orice şi refuza orice. Astfel, dacă există o mulţime care *nu poate* fi refuzată, procesul nu este divergent. Aceasta justifică formularea unei condiţii suficiente pentru nondivergenţă.

$NONDIV = (ref \neq A)$

Din fericire

$NONSTOP \Rightarrow NONDIV$

astfel că demonstrarea absenţei divergenţei nu presupune mai multă muncă decât demonstrarea absenţei blocajului.

3.7.1 Reguli

În următoarele reguli de inferenţă o specificaţie va fi scrisă în oricare din formele S, $S(ur)$, $S(ur,ref)$ potrivit necesităţilor. În toate cazurile se va înţelege că specificaţia va conţine *ur* şi *ref* printre variabilele sale libere.

După definiţia nedeterminismului, $(P \sqcap Q)$ se comportă ca P sau ca Q. De aceea, fiecare observaţie a comportării sale va fi o observaţie pentru P sau pentru Q sau pentru amândouă. De aceea această observaţie va fi descrisă prin specificarea lui P sau a lui Q sau a amândurora. Mai mult, inferenţa pentru nedeterminism are o formă excepţional de simplă.

L1 Dacă P **sat** S
şi Q **sat** T
atunci $(P \sqcap Q)$ **sat** $(S \vee T)$

Legea pentru *STOP* afirmă că acest proces nu face nimic şi refuză orice.

L2A $STOP_A$ **sat** $(ur = \langle\rangle \wedge ref \subseteq A)$

Deoarece refuzurile sunt totdeauna conţinute în alfabet (3.4.1 L8) clauza $ref \subseteq A$ poate fi omisă. Astfel, dacă omitem alfabetele cu totul (aşa cum o vom face în viitor), legea L2A este identică cu aceea pentru procese deterministe (1.10.2 L4A)

$STOP$ **sat** $ur = \langle\rangle$

Legea anterioară pentru prefix (1.10.2 L4B) este de asemenea validă dar nu este suficient de puternică pentru a dovedi că procesul nu se poate opri înaintea unei acţiuni iniţiale. Regula trebuie întărită prin menţionarea faptului că în starea iniţială, când $ur=\langle\rangle$, acţiunea iniţială nu poate fi refuzată.

L2B Dacă P **sat** $S(ur)$
atunci $(c \rightarrow P)$ **sat** $((ur=\langle\rangle \wedge c \notin ref) \vee (ur_0=c \wedge S(ur')))$

Legea pentru alegerea generală (1.10.2 L4) trebuie de asemenea întărită

L2 Dacă $\forall x \in B. P(x)$ **sat** $S(ur,x)$
atunci $(x{:}B \rightarrow P(x))$ **sat** $((ur=\langle\rangle \wedge (B \cap ref=\{\,\}) \vee (ur_0 \in B \wedge S(ur',ur_0))$

Legea pentru compunerea paralelă dată de 2.7 L1 este încă validă în condiţiile în care specificaţiile nu fac referire la mulţimi de refuzuri. Pentru a putea manipula corect refuzurile avem o lege ceva mai complicată.

L3 Dacă P **sat** $S(ur,ref)$
şi Q **sat** $T(ur,ref)$
şi nici P şi nici Q nu diverg
atunci $(P||Q)$ **sat** $(\exists X,Y. ref=(X \cup Y) \wedge S(ur \upharpoonright \alpha P, X) \wedge T(ur \upharpoonright \alpha Q, Y)\}$

Legea pentru schimbare de simbol necesită o modificare analoagă

L4 Dacă P **sat** $S(ur,ref)$
atunci $f(P)$ **sat** $S\,(f^{-1*}(ur), f^{-1}(ref))$ în condiţia că f este injectivă

Legea pentru □ este surprinzător de simplă

L5 Dacă P **sat** S
şi Q **sat** T
şi nici P şi nici Q nu diverg
atunci $(P \square Q)$ **sat** (**if** $ur=\langle\rangle$ **then** $(S \wedge T)$ **else** $(S \vee T)$)

Iniţial, când $ur=\langle\rangle$ o mulţime este refuzată de $(P \square Q)$ numai dacă este refuzată atât de P cât şi de Q. De aceea această mulţime trebuie să fie descrisă de *ambele* specificaţii. Mai mult, când $ur \neq \langle\rangle$, fiecare observaţie a lui $(P \square Q)$ trebuie să fie observaţie atât a lui P cât şi a lui Q şi de aceea trebuie să fie descrisă de una din specificaţiile lor (sau de ambele).

Legea pentru întreţesere nu necesită să menţionăm mulţimile de refuzuri.

L6 Dacă P **sat** $S(ur)$
și Q **sat** $T(ur)$
și nici P nici Q nu diverg
atunci $(P|||Q)$ **sat** $(\exists s,t.(ur \textit{ întrețese } (s,t) \wedge S(s) \wedge T(t)))$

Legea pentru mascare este mai complicată din necesitatea de a ne asigura că avem gardă în cazul divergenței

L7 Dacă P **sat** $(NODIV \wedge S(ur,ref))$
atunci $(P\backslash C)$ **sat** $\exists s.ur{=}s\upharpoonright(\alpha P - C) \wedge S(s, ref \cup C)$

unde $NODIV$ afirmă că numărul simbolurilor mascate ce pot să apară este mărginit de o funcție a simbolurilor nemascate ce au apărut

$$NODIV {=} \#(ur\upharpoonright C) \leq f(ur\upharpoonright(\alpha P - C))$$

unde f este o funcție definită pe mulțimea urmelor cu valori în numere naturale.

Clauza $ref \cup C$ din consecința legii L7 necesită explicații. Ea provine din faptul că $P\backslash C$ poate refuza o mulțime X numai când P poate refuza întreaga mulțime $X \cup C$, cu alte cuvinte X împreună cu *toate* evenimentele mascate. $P\backslash C$ nu poate refuza să interacționeze cu mediul său extern până ce n-a ajuns la o stare în care să refuze angajarea în orice viitoare activitate internă mascată. Acest fel de echitate este o caracteristică importantă a oricărei definiții de mascare așa cum s-a descris în paragraful 3.5.1.

Regula pentru recursivitate (1.10.2 L6) trebuie de asemenea întărită. Fie $S(n)$ un predicat conținând variabila n din gama numerelor naturale 0,1,2,3,.....

L8 Dacă $S(0)$
și $(X \textbf{ sat } S(n)) \Rightarrow (F(X) \textbf{ sat } S(n+1))$
atunci $(\mu X.F(X))$ **sat** $(\forall n.S(n))$

Această lege este bună chiar și pentru o recursivitate fără gardă cu toate că cea mai tare specificare ce poate fi dovedită pentru un astfel de proces este *true*.

3.8 Divergența

În capitolele anterioare am observat restricția ca ecuațiile care definesc un proces prin recursivitate să fie *cu gardă* (paragraful 1.1.2). Aceasta ne asigură

că ecuațiile au numai o singură soluție (1.3 L2). De asemenea restricția ne-a eliberat de a da o semnificație recursivității infinite

$\mu X.X$

Din nefericire, introducerea mascării (paragraful 3.5) înseamnă că o recursivitate aparent cu gardă poate să nu fie constructivă. De exemplu să considerăm ecuația

$X=c\to X\backslash\{c\})_{+\{c\}}$

Aceasta are ca soluții atât ($c\to STOP$) cât și ($c\to a\to STOP$), care pot fi dovedite prin substituție.

Mai mult, orice ecuație recursivă care implică operatorul de mascare aplicat recursivității este potențial *fără gardă* și pasibilă de a avea mai mult de o soluție. Care soluție va fi cea corectă? Afirmăm că soluția corectă va fi cea mai puțin deterministă din cauză că aceasta permite o alegere nedeterministă între celelalte soluții. Cu această explicație putem scoate cu totul restricția ca recursivitățile să fie cu gardă și putem da o semnificație (posibil nedeterministă) *oricărei* expresii de forma $\mu X.F(X)$ unde F este definită în termenii oricărui operator introdus în această carte (cu excepția lui /) și respectând toate constrângerile de alfabet.

Pentru a explica această semnificație vom trata întâi cel mai simplu caz, care (cum se întâmplă adesea) este cel mai defavorabil, recursivitatea infinită

$\mu X.X$

Orice proces este o soluție a ecuației recursive

$X=X$

Prin urmare, $\mu X.X$ se poate comporta în definitiv ca oricare proces. Este cel mai nedeterminist din toate procesele, cel mai puțin previzibil, cel mai puțin controlabil, pe scurt cel mai rău. Să-i dăm un nume potrivit și să-l definim

$CHAOS_A=\mu X{:}A.X$

Un caz ceva mai bun este recursivitatea

$\mu X.(c\to(X\backslash\{c\}))_{+\{c\}}=c\to CHAOS$

Acesta este un proces diferit de *CHAOS* din cauză că se poate angaja cel puțin în evenimentul inițial c, înainte de a intra în *CHAOS*.

În afară de semnificația recursivității infinite, *CHAOS* este de asemenea rezultatul angajării unui proces într-o succesiune infinită de evenimente consecutive ascunse. Cel mai simplu și cel mai rău caz este un proces imediat divergent menționat anterior la sfârșitul paragrafului 3.5.1

$$(\mu X{:}A.(c \rightarrow X)) \backslash \{c\} = \mu X{:}(A-\{c\}).(c \rightarrow X) \backslash \{c\}$$

$$= \mu X{:}A-\{c\}.(X \backslash \{c\}) \qquad \text{din 3.5.1.L5}$$

$$= \mu X{:}A-\{c\}.X$$

$$= CHAOS_{A-\{c\}} \qquad \text{definiția } CHAOS$$

3.8.1 Legi

Deoarece *CHAOS* este cel mai nedeterminist proces nu poate fi schimbat prin aplicarea alegerii nedeterministe fiind de aceea un element neutru pentru $\sqcap$

L1 $P \sqcap CHAOS = CHAOS$

O funcție de procese care produce *CHAOS* când unul din argumentele sale este *CHAOS* se spune că este *strictă*. Legea de mai sus (plus simetria) afirmă că $\sqcap$ este strictă. *CHAOS* este un astfel de proces îngrozitor încât orice proces care este definit în termeni de *CHAOS* este el însuși egal cu *CHAOS*

L2 Următoarele operații sunt stricte
$/s$, $\|$, f, $\square$, $\backslash C$, $|||$ și μX

Totuși prefixul nu este strict.

L3 $CHAOS \neq (a \rightarrow CHAOS)$

din cauză că partea dreaptă se poate baza pe execuția lui a înainte de a deveni complet necontrolabilă.

Așa cum s-a mai spus, *CHAOS* este cel mai neprevizibil și cel mai necontrolabil dintre procese. Poate face orice, poate refuza orice

L4 $urme(CHAOS_A) = A^*$
L5 $refuzuri(CHAOS_A) =$ toate submulțimile lui A

3.8.2 Divergențe

O *divergență* a unui proces este prin definiție orice urmă a unui proces după parcurgerea căreia procesul se comportă haotic. Mulțimea tuturor divergențelor este definită

$divergențe(P)=\{s \mid s \in urme(P) \wedge (P/s)=CHAOS_{\alpha P}\}$

Rezultă imediat că

L1 $divergențe(P) \subseteq urme(P)$

Din cauză că /t este strictă

$CHAOS/t=CHAOS$

Urmează că divergenţele unui proces sunt închise la concatenare în sensul că

L2 $s \in divergențe(P) \wedge t \in (\alpha P)^* \Rightarrow (s\char`^t) \in divergențe(P)$

Deoarece $CHAOS_A$ poate refuza orice submulţime a alfabetului său A

L3 $s \in divergențe(P) \wedge X \subseteq \alpha P \Rightarrow X \in refuzuri(P/s)$

Cele trei legi date mai sus pun în evidenţă proprietăţile generale de divergenţă ale oricărui proces. Următoarele legi arată cum divergenţele proceselor compuse sunt determinate de divergenţele şi urmele componentelor lor. Mai întâi, procesul *STOP* nu diverge niciodată

L4 $divergențe(STOP)=\{\langle\rangle\}$

La cealaltă extremă, orice urmă a lui *CHAOS* ne conduce la *CHAOS*

L5 $divergențe(CHAOS_A)=A^*$

Un proces definit prin posibilitatea alegerii nu diverge în primul său pas. Mai mult, divergenţele sale sunt determinate de ce se întâmplă după primul său pas.

L6 $divergențe(x{:}B \rightarrow P(x))=\{\langle x\rangle\char`^s \mid x \in B \wedge s \in divergențe(P(x))\}$

Orice divergenţă a lui P este de asemenea o divergenţă a lui $(P \sqcap Q)$ şi a lui $(P \square Q)$

L7 $divergențe(P \sqcap Q)=divergențe(P \square Q)$
$=divergențe(P) \cup divergențe(Q)$

Deoarece operatorul || este strict, o divergenţă a lui $(P||Q)$ porneşte cu o urmă a activităţii nedivergente atât a lui P cât şi a lui Q care ne conduce la divergenţa atât a lui P cât şi a lui Q (sau a ambelor).

L8 $divergențe(P||Q)=$
$\{s\hat{}t \mid t\in(\alpha P\cup\alpha Q)^* \wedge((s\upharpoonright\alpha P\in divergențe(P)\wedge s\upharpoonright\alpha Q\in urme(Q))$
$\vee(s\upharpoonright\alpha P\in urme(P)\wedge s\upharpoonright\alpha Q\in divergențe(Q)))\}$

O explicaţie similară se aplică lui |||

L9 $divergențe(P|||Q)=$
$\{u \mid \exists s,t.\ u\ \textit{întrețese}(s,t)\wedge((s\in divergențe(P)\wedge t\in urme(Q))$
$\vee(s\in urme(P)\wedge t\in divergențe(Q)))\}$

Divergenţele unui proces rezultate din mascare includ urme provenite din divergenţele originale plus acelea rezultate din încercarea de a masca o secvenţă infinită de simboluri.

L10 $divergențe(P\backslash Q)=\{(s\upharpoonright(\alpha P-C))\hat{}t \mid t\in(\alpha P-C)^*$
$\wedge(s\in divergențe(P)$
$\vee(\forall n.\exists u\in C^*.\#u>n\wedge(s\hat{}u)\in urme(P)))\}$

Un proces definit prin schimbare de simbol diverge numai dacă argumentul său diverge.

L11 $divergențe(f(P))=\{f^*(s) \mid s\in divergențe(P)\}$
în condiţiile când f este injectivă

Este totuşi neplăcut de a dedica atât de multă atenţie divergenţei când ea este ceva ce *nu* dorim să apară. Din nefericire, ea pare a fi o consecinţă inevitabilă a oricărei metode eficiente sau chiar procedurale de implementare. Divergenţa poate proveni din mascare sau recursivitate fără gardă. De aceea o parte a muncii unui proiectant de sistem constă în a dovedi că în proiectul său divergenţa nu va apare. Astfel pentru a dovedi că ceva nu poate să apară avem nevoie de o teorie matematică în care acel ceva să fie posibil.

3.9. Teoria matematică a proceselor nedeterministe

Legile date în acest capitol sunt evident mai complicate decât cele din capitolele anterioare. Justificările explicative şi exemplele sunt prin urmare mai puţin convingătoare. Este de aceea important să construim o definiţie mate-

matică adecvată a conceptului de proces nedeterminist şi să dovedim corectitudinea legilor din definiţiile operatorilor.

Ca şi în paragraful 2.8.1, un model matematic se bazează pe proprietăţile direct sau indirect relevante ale unui proces. Acestea includ desigur alfabetul şi urmele sale, dar pentru un proces nedeterminist există de asemenea refuzurile sale (paragraful 3.4) şi divergenţele (paragraful 3.8). În plus faţă de refuzurile din primul pas al unui proces P, este necesar de a se lua totodată în calcul ce poate refuza P după angajarea într-o urmă arbitrară s a comportării sale. De aceea definim ca *eşecuri* ale unui proces P o relaţie (mulţime de perechi)

$$eşecuri(P)=\{(s,X) \mid s \in urme(P) \wedge X \in refuzuri(P/s)\}$$

Dacă (s,X) este un eşec a lui P, aceasta înseamnă că P se poate angaja în orice secvenţă de evenimente înregistrată prin s şi apoi să refuze orice în ciuda faptului că mediul este pregătit să se angajeze în oricare din evenimentele lui X. Eşecurile unui proces informează mai mult asupra comportării procesului decât urmele sau refuzurile sale care pot fi amândouă definite în termenii eşecurilor

$$\begin{aligned} urme(P) &= \{s \mid \exists X.(s,X) \in eşecuri(P)\} \\ &= domeniu(eşecuri(P)) \\ refuzuri(P) &= \{X \mid (\langle\rangle,X) \in eşecuri(P)\} \end{aligned}$$

Feluritele proprietăţi ale urmelor (1.8.1 L6, L7, L8) şi refuzurilor (3.4 L8, L9, L10, L11) pot fi uşor reformulate în termenii eşecurilor (vezi condiţiile C0, C1, C2, C3 după definiţia D0 de mai jos).

Acum suntem gata pentru decizia importantă ca un proces să fie definit de trei mulţimi specificând alfabetul, eşecurile şi divergenţele. Reciproc, orice trei mulţimi care satisfac anumite condiţii definesc unic un proces. Vom defini mulţimea părţilor lui A ca mulţimea tuturor submulţimilor sale

$$\mathbf{P}_A=\{X \mid X \subseteq A\}$$

D0 Un proces este un triplet (A,F,D)

unde A este orice mulţime de simboluri (pentru simplitate finită)

F este o relaţie între A^* şi $\mathbf{P}_A$

D este o submulţime a lui A^*

în cazul când satisfac următoarele condiţii

C0 $(\langle\rangle,\{\}) \in F$

C1 $(s^\frown t,X) \in F \Rightarrow (s,\{\}) \in F$

C2 $(s,Y)\in F \wedge X\subseteq Y \Rightarrow (s,X)\in F$
C3 $(s,X)\in F \wedge x\in A \Rightarrow (s,X\cup\{x\})\in F \vee (s\hat{}\langle x\rangle,\{\})\in F$
C4 $D\subseteq domeniu(F)$
C5 $s\in D \wedge t\in A^{*} \Rightarrow s\hat{}t\in D$
C6 $s\in D \wedge X\subseteq A \Rightarrow (s,X)\in F$

(ultimele trei condiţii reflectă legile 3.8.2 L1, L2, L3).
Cel mai simplu proces care satisface această definiţie este şi cel mai rău

D1 $CHAOS_A=(A,(A^{*}\times \mathbf{P}_A),A^{*})$

unde $A^{*}\times \mathbf{P}_A$ este produs cartezian

$$\{(s,X)\mid s\in A^{*} \wedge X\in \mathbf{P}_A\}$$

Acest proces este cel mai mare proces cu alfabetul A, deoarece orice membru al lui A^{*} este atât o urmă cât şi o divergenţă şi orice submulţime a lui A este un refuz după toate urmele.

Alt proces simplu este definit

D2 $STOP_A=(A,\{\langle\rangle\}\times \mathbf{P}_A,\{\})$

Acest proces nu face nimic, poate refuza orice şi nu are divergenţe.

Vom defini operatori pe mulţimea proceselor arătând cum cele trei mulţimi ale rezultatului provin din cele ale operanzilor. Desigur este necesar de arătat că rezultatul operaţiei satisface cele şase condiţii din D0. Aceasta se bazează pe presupunerea că operanzii permit acest lucru.

Cea mai simplă operaţie de definit este *sau* nedeterminist ($\sqcap$). Ca mulţi alţi operatori el este definit numai pentru operanzi cu acelaşi alfabet.

D3 $(A,F1,D1)\sqcap(A,F2,D2) = (A,F1\cup F2,D1\cup D2)$

Procesul rezultant poate eşua sau diverge în toate cazurile în care unul din cei doi operanzi poate face aceasta. Legile 3.2.1 L1, L2, L3 sunt consecinţe directe ale acestei definiţii. Definiţiile pentru ceilalţi operatori pot fi date similar dar apare mai elegant de a scrie definiţii separate pentru alfabete, eşecuri şi divergenţe. Definiţiile pentru divergenţe au fost date în paragraful 3.8.2, deci rămân de definit alfabetele şi eşecurile.

D4 Dacă $\alpha P(x)=A$ pentru toţi x
şi $B\subseteq A$
atunci $\alpha(x:B\rightarrow P(x))=A$

D5 $\alpha(P\|Q)=(\alpha P\cup\alpha Q)$

D6 $\alpha(f(P))=f(\alpha P)$

D7 $\alpha(P\Box Q)=\alpha(P|||Q)=\alpha P$ când $\alpha P=\alpha Q$

D8 $\alpha(P\backslash C)=\alpha P-C$

D9 $eşecuri(x{:}B\rightarrow P(x))=\{\langle\rangle,X\mid X\subseteq(\alpha P-B)\}$
$\cup\{\langle x\rangle\hat{\ }s,X\mid x\in B\wedge(s,X)\in eşecuri\ (P(x))\}$

D10 $eşecuri(P\|Q)=\{s,(X\cup Y)\mid s\in(\alpha P\cup\alpha Q)^{*}$
$\wedge(s\upharpoonright\alpha P,X)\in eşecuri(P)$
$\wedge(s\upharpoonright\alpha Q,Y)\in eşecuri(Q)\}$
$\cup\{s,X\mid s\in divergenţe(P\|Q)\}$

D11 $eşecuri(f(P))=\{f^{*}(s),f(X)\mid(s,X)\in eşecuri\ (P)\}$

D12 $eşecuri(P\Box Q)=\{s,X\mid(s,X)\in eşecuri(P)\cap eşecuri(Q)$
$\vee(s\neq\langle\rangle\wedge(s,X)\in(eşecuri(P)\cup eşecuri(Q)))\}$
$\cup\{s,X\mid s\in divergenţe(P\Box Q)\}$

D13 $eşecuri(P|||Q)=\{s,X\mid\exists t,u.s\ întreţese\ (t,u)$
$\wedge(t,X)\in eşecuri(P)$
$\wedge(u,X)\in eşecuri(Q)\}$
$\cup\{s,X\mid s\in divergenţe(P|||Q)\}$

D14 $eşecuri(P\backslash C)=\{s\upharpoonright(\alpha P-C),X\mid(s,X\cup C)\in eşecuri(P)\}$
$\cup\{s,X\mid s\in divergenţe(P\backslash C)\}$

Explicarea acestor legi poate fi făcută funcţie de urmele şi refuzurile corespunzătoare împreună cu legile pentru $\diagup$.

Rămâne de a da o definiţie proceselor definite recursiv cu ajutorul lui μ. Metoda se bazează pe aceeaşi teoremă de punct fix ca în paragraful 2.8.2 cu excepţia faptului că definiţia ordinii este diferită.

D15 $(A,F1,D1)\sqsubseteq(A,F2,D2)\equiv(F2\subseteq F1\wedge D2\subseteq D1)$

$P\sqsubseteq Q$ semnifică faptul că Q este egal cu P sau chiar mai mult în sensul divergenţei şi eşecului (mai puţin probabil să diveargă şi să eşueze). Q este mai previzibil şi mai controlabil decât P, din cauză că dacă Q poate face ceva nedorit, P poate face de asemenea, iar dacă Q poate refuza ceva, atunci P poate de asemenea. *CHAOS* poate face orice oricând şi poate refuza orice oricând. Conform numelui, el este cel mai puţin previzibil şi controlabil dintre procese sau pe scurt cel mai rău.

L1 $CHAOS\sqsubseteq P$

Această ordonare este clar o ordine parţială. De fapt este o ordine parţială completă cu o operaţie de limită definită în termenii intersecţiilor lanţurilor descrescătoare de eşecuri şi divergenţe.

D16 $\bigsqcup_{n\geq 0} (A,F_n,D_n) = (A, \bigcap_{n\geq 0} F_n, \bigcap_{n\geq 0} D_n)$

În condiţiile când $(\forall n\geq 0.F_{n+1}\subseteq F_n \wedge D_{n+1}\subseteq D_n)$.

Operatorul μ este definit la fel ca pentru procese deterministe (2.8.2 L7) cu excepţia unei diferenţe în definiţia ordinii care necesită ca *CHAOS* să fie folosit în locul lui *STOP*.

D17 $\mu X{:}A.F(X) = \bigsqcup_{i\geq 0} F^i(CHAOS_A)$

Demonstraţia că aceasta este soluţia (de fapt cea mai nedeterministă soluţie) unei ecuaţii date este aceeaşi cu cea dată în paragraful 2.8.2.

Ca şi atunci, validitatea demonstraţiei depinde critic de faptul că toţi operatorii folosiţi în partea dreaptă a recursivităţii trebuie să fie continui conform ordinii. Din fericire, toţi operatorii definiţi în această carte (cu excepţia lui /), sunt continui şi astfel orice formulă construită cu ei. În cazul operatorului de mascare, continuitatea a fost una din cerinţele principale pentru tratarea complexă a divergenţei.

4 Comunicația

4.1 Introducere

În capitolele anterioare s-a introdus și ilustrat conceptul general de eveniment ca o acțiune fără durată a cărui apariție necesită participarea simultană a cel puțin două procese descrise independent. În acest capitol ne vom concentra asupra unei clase speciale de evenimente cunoscute ca evenimente de *comunicații*. O comunicație este un eveniment descris printr-o pereche

$$c.v$$

unde c este numele canalului pe care se face comunicația și v este valoarea mesajului transferat. Exemple ale acestei convenții au fost date deja în *COPIEBIT* (1.1.3 X7) și *LANȚ2* (2.6 X4).

Mulțimea tuturor mesajelor pe care P le poate transfera pe canalul c este definită

$$\alpha c(P)=\{v \mid c.v\in\alpha P\}$$

De asemenea definim funcțiile care extrag componenta de canal și mesaj dintr-o comunicație

$$canal(c.v)=c, \quad mesaj(c.v)=v$$

Toate operațiile introduse în acest capitol pot fi definite prin termenii conceptelor intuitive introduse în capitolele anterioare și multe din legi sunt cazuri particulare ale unor legi familiare anterioare. Motivul pentru introducerea notațiilor speciale constă în faptul că sunt sugestive în utilizare și în implementare. De asemenea impunerea unor restricții notaționale permite utilizarea unor metode de raționament mai puternice.

4.2 Intrare şi ieşire

Fie v un membru din $\alpha c(P)$. Un proces care întâi transmite v pe canalul c şi apoi se comportă ca P, este definit

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

Singurul eveniment în care acest proces este pregătit iniţial să se angajeze este evenimentul de comunicaţie $c.v$.

Un proces care este iniţial pregătit să primească orice valoare x transferabilă pe canalul c, şi apoi să se comporte ca $P(x)$ este definit

$$(c?x \rightarrow P(x)) = (y{:}\{y \mid canal(y) = c\} \rightarrow P(mesaj(y)))$$

Exemple

X0 Folosind noile definiţii pentru intrare şi ieşire putem rescrie 1.1.3 X7

$$COPIEBIT = \mu X.(in?x \rightarrow (out!x \rightarrow X))$$

unde $\alpha in(COPIEBIT) = \alpha out(COPIEBIT) = \{0,1\}$ □

Se observă convenţia ca aceste canale să fie folosite pentru comunicaţii unidirecţionale şi numai între două procese. Un canal care este folosit numai pentru transferuri de ieşire într-un proces se va chema canal de ieşire (transmisie) a acelui proces, iar unul folosit numai pentru transferuri de intrare, se va chema canal de intrare (recepţie). În ambele cazuri, vom spune într-un sens mai larg că numele unui canal este un membru al alfabetului procesului.

Când desenăm o diagramă de conexiune (paragraful 2.4) a unui proces, canalele sunt desenate cu săgeţi în direcţia corespunzătoare sensului de transfer şi etichetate cu numele canalului (fig. 4.1).

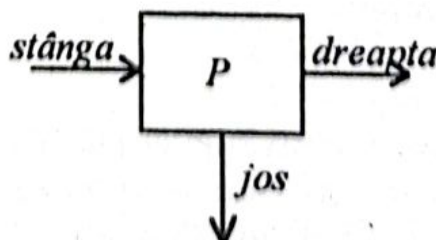


Figura 4.1

Fie P şi Q procese şi c un canal de ieşire pentru P şi intrare pentru Q. Când P şi Q sunt compuse concurent într-un sistem $(P \| Q)$, comunicaţia va

apare pe canalul c de fiecare dată când P emite un mesaj şi Q simultan recepţionează acel mesaj. Un proces care emite un mesaj precizează (specifică unic) valoarea lui, iar un proces care recepţionează este pregătit să accepte orice valoare comunicabilă. Evenimentul care va apare de fapt este comunicaţia $c.v$ unde v este valoarea specificată în procesul care transmite. Aceasta necesită constrângerea evidentă ca acel canal c să aibă acelaşi alfabet la ambele capete

$$\alpha c(P)=\alpha c(Q)$$

Pe viitor vom presupune îndeplinită această constrângere şi unde nu se poate naşte nici o confuzie vom scrie αc pentru $\alpha c(P)$. Un exemplu pentru funcţionarea acestui model de comunicaţie a fost dat în *LANŢ2* (2.6 X4). Exemple şi mai interesante vor fi date în paragraful 4.3 şi următoarele.

În general, mesajul ce va fi transmis de un proces este specificat cu ajutorul unei expresii conţinând variabile cărora li s-a asignat o valoare într-o recepţie anterioară, aşa cum se arată în exemplele următoare.

Exemple

X1 Un proces care copie imediat orice mesaj pe care îl recepţionează pe canalul din stânga, transmiţându-l prin canalul din dreapta

$$\alpha stânga(COPIE)=\alpha dreapta(COPIE)$$
$$COPIE=\mu X.(stânga?x\rightarrow dreapta!x\rightarrow X)$$

Dacă $\alpha stânga=\{0,1\}$, *COPIE* este aproape identic cu *COPIEBIT* (1.1.3 X7)

□

X2 Un proces asemănător lui *COPIE* dar care dublează valoarea la ieşire

$$\alpha stânga=\alpha dreapta=\mathbf{N}$$
$$DUBLU=\mu X.(stânga?x\rightarrow dreapta!(x+x)\rightarrow X)$$

□

X3 O cartelă este o secvenţă de 80 de caractere care pot fi citite ca un singur mesaj prin canalul stâng al unui proces. Un proces care citeşte cartele şi transmite caracterele lor unul câte unul

$$\alpha stânga=\{s\mid s\in\alpha dreapta^{*}\wedge \#s=80\}$$

$$DESPACHET=P_{\langle\rangle}$$

unde $P_{\diamond}$=*stânga?s*→P_s

şi $P_{\langle x\rangle}$=*dreapta!x*→$P_{\diamond}$ □

$P_{\langle x\rangle\wedge s}$=*dreapta!x*→$P_s$

X4 Un proces recepţionează caracterele prin canalul din stânga unul câte unul şi le asamblează în linii de câte 125 caractere lungime. Fiecare linie completă este transmisă apoi prin canalul din dreapta ca un mesaj-vector

α*dreapta*={*s* | *s*∈α*stânga** ∧#*s*=125}

ÎMPACHET=$P_{\diamond}$

unde P_s=*dreapta!s*→$P_{\diamond}$ dacă #*s*=125

şi P_s=*stânga?x*→$P_{s\wedge\langle x\rangle}$ dacă #*s*<125

Aici P_s descrie comportarea unui proces care recepţionează şi împachetează caracterele în secvenţa *s*, ele urmând să fie transmise când linia este completă. □

X5 Un proces care tot ce recepţionează prin canalul din stânga copie transmiţând prin canalul din dreapta, exceptând perechile de asteriscuri consecutive înlocuite cu un singur semn "↑"

α*stânga*=α*dreapta*–{"↑"}

ELIMIN=μ*X*. *stânga?x*→

if *x*≠"*" **then** (*dreapta!x*→*X*)

else *stânga?y*→(**if** *y*="*" **then** (*dreapta*!"↑"→*X*)

else (*dreapta*!"*"→*dreapta!y*→*X*)) □

Un proces trebuie să fie pregătit iniţial să comunice pe oricare dintr-o mulţime de canale, alegerea acestora lăsând-o altor procese cu care este conectat. Pentru aceasta, adaptăm notaţia alegerii introdusă în cap. 1. Dacă *c* şi *d* sunt nume de canale distincte

(*c?x*→*P*(*x*) | *d?y*→*Q*(*y*))

semnifică un proces care iniţial recepţionează *x* pe *c* şi apoi se comportă ca *P*(*x*) sau iniţial recepţionează *y* pe canalul *d* şi apoi se comportă ca *Q*(*y*). Alegerea este determinată de oricare dintre ieşirile corespunzătoare care este gata, aşa cum se arată mai jos.

Deoarece am decis să facem abstracţie de timingul evenimentelor şi de viteza proceselor care se angajează în ele, ultima exprimare din paragraful precedent necesită explicaţii. Considerăm cazul când canalele *c* şi *d* sunt ca-

nale de ieşire ale altor două procese separate care sunt independente în sensul că nu comunică direct sau indirect între ele. Acțiunile acestor două procese sunt de aceea întrețesute. Astfel, dacă un proces progresează în sensul transmiterii unui mesaj pe canalul *c* şi celălalt progresează în sensul transmiterii unui mesaj pe *d*, nu este determinat care dintre ele va face primul transmisia. Un implementator va dori, într-un fel sau altul, să rezolve acest nedeterminism de exemplu în favoarea primei ieşiri ce devine disponibilă. Această politică protejează de asemenea împotriva blocajului care rezultă dacă a doua ieşire nu va apare niciodată sau dacă apare numai după prima, ca în cazul când ambele canale *c* şi *d* aparțin aceluiaşi proces concurent care transmite pe unul şi apoi pe celălalt canal

$(c!2 \to d!4 \to P)$

De aceea prezența alternativei intrărilor nu numai că protejează împotriva blocajului dar de asemenea realizează o eficiență sporită şi reduce timpii de acces la comunicațiile propuse. Un călător care aşteaptă autobuzul 28 de exemplu, va trebui să aştepte mai mult decât cel care este pregătit să călătorească şi cu autobuzul 9, şi cu 28, oricare ar sosi primul în stație. Având în vedere sosirile aleatoare, călătorul care are cele două alternative va aştepta, paradoxal, jumătate din timp. Astfel, a aştepta primul dintre mai multe evenimente posibile este singura cale de a realiza dezideratul de mai sus. A cumpăra computere mai rapide este inutil.

X6 Un proces care recepționează pe oricare două canale *stânga*1 sau *stânga*2 şi imediat transmite mesajul în dreapta

$\alpha stânga1 = \alpha stânga2 = \alpha dreapta$
$UNIFICĂ = (stânga1?x \to dreapta!x \to UNIFICĂ$
$\quad | stânga2?x \to dreapta!x \to UNIFICĂ)$

Ieşirea acestui proces este întrețeserea mesajelor de intrare de la canalele *stânga*1 şi *stânga*2. □

X7 Un proces care totdeauna este pregătit să primească o valoare prin stânga sau să transmită în dreapta valoarea pe care tocmai a recepționat-o

$\alpha stânga = \alpha dreapta$
$VAR = stânga?x \to VAR_x$

unde $VAR_x = (stânga?y \to VAR_y$
$\quad | dreapta?x \to VAR_x)$

Aici procesul VAR_x se comportă ca o variabilă de program cu valoarea curentă x. Valori noi îi sunt asignate prin comunicaţiile primite de la canalul său stâng şi valoarea sa curentă este accesată prin comunicarea făcută prin canalul din dreapta. Dacă $\alpha st\hat{a}nga=\{0,1\}$, comportarea lui VAR este aproape identică cu cea a lui $BOOL$ (2.6 X5). □

X8 Un proces care primeşte de la canalele *sus* şi *stânga* şi emite prin canalul *jos* o funcţie de intrări înainte de a repeta acţiunea

$$NOD(\nu)=\mu X.(sus?sum\breve{a}\rightarrow st\hat{a}nga?prod\rightarrow jos!(sum\breve{a}+\nu\times prod)\rightarrow X)$$ □

X9 Un proces care este totdeauna gata să recepţioneze un mesaj prin stânga şi să transmită prin dreapta primul mesaj primit dar netransmis încă

$$BUFFER=P_{\langle\rangle}$$

unde $P_{\langle\rangle}=st\hat{a}nga?x\rightarrow P_{\langle x\rangle}$

şi $P_{\langle x\rangle\frown s}=(st\hat{a}nga?y\rightarrow P_{\langle x\rangle\frown s\frown\langle y\rangle}$

$\quad|dreapta!x\rightarrow P_s)$

BUFFER se comportă ca o coadă. Mesajele se alătură capătului din stânga al cozii şi-o părăsesc prin partea dreaptă în aceeaşi ordine în care s-au alăturat dar după o întârziere posibilă, timp în care alte mesaje se pot alătura cozii. □

X10 Un proces se comportă ca o stivă de mesaje. Când este goală, răspunde cu semnalul *gol*. Totdeauna este gata de a primi un nou mesaj prin canalul din stânga şi de a-l pune pe stivă. Ori de câte ori nu este goală, procesul stivă este gata de a transmite prin canalul din dreapta şi a înlocui vârful său

$$STIV\breve{A}=P_{\langle\rangle}$$

unde $P_{\langle\rangle}=(gol\rightarrow P_{\langle\rangle}|st\hat{a}nga?x\rightarrow P_{\langle x\rangle})$

şi $\quad P_{\langle x\rangle\frown s}=(dreapta!x\rightarrow P_s|st\hat{a}nga?y\rightarrow P_{\langle y\rangle\frown\langle x\rangle\frown s})$

Acest proces este foarte asemănător cu procesul precedent, cu excepţia faptului că atunci când stiva este goală procesul aferent participă la evenimentul *gol* şi de asemenea noile mesaje sosite sunt adăugate la acelaşi capăt al secvenţei depuse din care se şi ia când este cazul. Astfel, dacă y este noul mesaj de in-

trare şi *x* este mesajul curent gata pentru transmis, procesul *STIVĂ* reţine <y>^<x>^*s* pe când *BUFFER* reţine <x>^*s*^<y>. □

4.2.1 Implementare

Într-o implementare LISP a proceselor comunicante evenimentul *c.ν* este natural reprezentat de perechea cu punct (*c.ν*) şi este construită cu

cons("*c*,ν).

Comenzile de recepţie şi de transmisie sunt convenabil implementate ca funcţii care întâi preiau numele canalului ca argument. Dacă procesul nu este pregătit să comunice pe canal, dă răspunsul "*BLIP*. Valoarea efectivă transmisă prin comunicaţie este tratată separat în următorul pas, cum se arată mai jos.

Dacă *Q* este comanda de intrare

$(c?x \rightarrow Q(x))$

atunci *Q*("*c*)≠"*BLIP*. În afară de aceasta, rezultatul ei este o funcţie care aşteaptă valoarea de intrare *x* ca argument şi generează ca rezultat procesul *Q*(*x*). Astfel *Q* este implementat prin apelul funcţiei LISP

recep("*c*,λ*x*.*Q*(*x*))

care este definită

recep(*c*,*F*)=λ*y*. **if** *y*≠*c* **then** "*BLIP* **else** *F*

Urmează că expresia *Q*/<*c.ν*> este reprezentat în LISP de *Q*("*c*)(ν) în condiţiile când <*c.ν*> este o urmă a lui *Q*.

Dacă *P* este comanda de ieşire

$(c!\nu \rightarrow P')$

atunci *P*("*c*)≠*BLIP*. Altfel, rezultatul său este perechea *cons*(ν,*P'*). Astfel, *P* este implementat prin apelul funcţiei *LISP*

transmit ("*c*,ν,*P'*)

care este definită

$transmit(c,\nu,P)=\lambda y.$ **if** $y\neq c$ **then** "$BLIP$ **else** $cons(\nu,P)$

Urmează că $\nu=car(P("c))$ şi $P/<c.\nu>$ este reprezentat în LISP prin $cdr(P("c))$, în condiţiile când $<c.\nu>$ este o urmă a lui P.

În teorie, dacă αc este finit, este posibil să tratăm $c.\nu$ ca un singur eveniment transmis ca un parametru comenzilor de recepţie şi de transmisie. Dar aceasta ar fi foarte ineficient deoarece singura cale de a identifica ce valoare este transmisia ar fi să testăm dacă $P(c.\nu)\neq$"$BLIP$ pentru toate valorile ν în αc până când este găsită cea corectă. Una din justificările de a introduce notaţii separate pentru recepţie şi transmisie este de a încuraja şi permite metode de implementare care sunt mult mai eficiente. Dezavantajul este că implementarea aproape a toţi ceilalţi operatori trebuie să fie privită prin prisma acestei optimizări.

Exemple

X1 *COPIE=LABEL X. recep("stânga,λx.transmit("dreapta,x,X))* □

X2 *ÎMPACHET=P(NIL)*

unde *P=LABEL X.*

$\lambda s.$ **if** $lung(s)=125$ **then** *transmit("dreapta,s,X(NIL))*
else *recep("stânga,λx.X(append(s,cons(x,NIL))))* □

4.2.2 Specificaţii

În specificarea comportării unui proces comunicant este convenabil să descriem separat secvenţele mesajelor care trec de-a lungul fiecărui canal. Dacă c este un nume de canal, definim (vezi 1.9.6)

$$tr\downarrow c=mesaj^{*}(tr\restriction \alpha c)$$

Este convenabil să omitem $tr\downarrow$ şi să scriem $dreapta\leq stânga$ în loc de $tr\downarrow dreapta\leq tr\downarrow stânga$.

Altă definiţie utilă plasează o margine inferioară lungimii prefixului

$$s\overset{n}{\leq} t=(s\leq t\wedge \#t\leq \#s+n)$$

Aceasta înseamnă că s este un prefix al lui t, mai puţin n termeni. Următoarele legi sunt evidente şi utile

$$s \overset{0}{\leq} t \equiv (s = t)$$

$$s \overset{n}{\leq} t \wedge t \overset{m}{\leq} u \Rightarrow s \overset{n+m}{\leq} u$$

$$s \leq t \equiv \exists n . s \overset{n}{\leq} t$$

Exemple

X1 *COPIE* **sat** $dreapta \overset{1}{\leq} stânga$ □

X2 *DUBLU* **sat** $dreapta \overset{1}{\leq} dublu^{*}(stânga)$ □

X3 *DESPACHET* **sat** $dreapta \leq \wedge / stânga$
unde $\wedge / <s_0, s_1, \ldots, s_{n-1}> = s_0 \wedge s_1 \wedge \ldots \wedge s_{n-1}$

Aici specificarea obligă ca secvenţa transmisă prin partea dreaptă să fie obţinută din concatenarea succesivă a secvenţelor recepţionate în stânga. □

X4 *ÎMPACHET* **sat** $((\wedge / dreapta \overset{125}{\leq} stânga) \wedge (\#^{*} dreapta) \in \{125\}^{*})$

Această specificare afirmă că orice element transmis în dreapta este el însuşi o secvenţă de 125 caractere şi concatenarea tuturor acestor secvenţe este o subsecvenţă a ceea ce s-a recepţionat în stânga. □

Dacă ⊕ este un operator binar este convenabil să-l aplicăm distributiv elementelor corespondente ale celor două secvenţe. Lungimea secvenţei rezultante este egală cu aceea a celui mai scurt operand

$s \oplus t = <>$ dacă $s = <>$ sau $t = <>$

$= (s_0 \oplus t_0) \wedge (s' \oplus t')$ altfel

Mai clar $(s \oplus t)[i] = s[i] \oplus t[i]$ pentru $i < \min(\#s, \#t)$

şi $s \overset{n}{\leq} t \Rightarrow (s \oplus u \overset{n}{\leq} t \oplus u) \wedge (u \oplus s \overset{n}{\leq} u \oplus t)$

X5 Şirul lui Fibonacci

<1,1,2,3,5,8,...>

este definit prin relaţia de recurenţă:

$fib[0] = fib[1] = 1$
$fib[i+2] = fib[i+1] + fib[i]$

A doua relaţie poate fi rescrisă folosind operatorul ' care deplasează secvenţa de numere o poziţie la stânga

$$fib'' = fib' + fib$$

Definiţia originală a şirului lui Fibonacci se poate rescrie într-o formă mai complicată folosind indexul

$$fib''[i] = (fib' + fib)[i]$$

de aceea $fib'[i+1] = fib'[i] + fib[i]$ (1.9.4 L1)

şi $fib[i+2] = fib[i+1] + fib[i]$.

Altă semnificaţie a ecuaţiei se poate face printr-o descriere a sumei infinite, unde deplasarea spre stânga o poziţie este arătată clar

```
1,1,2,3,5,...      fib
 / / / /
1,2,3,5,...       +fib'
 / / /
2,3,5,...         =fib''
```

În situaţia de mai sus, *fib* este privită ca o secvenţă infinită. Dacă s este o secvenţă iniţială finită a lui *fib* (cu $\#s \geq 2$), atunci în loc de ecuaţie avem inegalitatea

$$s'' \leq s' + s$$

Această formulare poate fi folosită pentru a specifica un proces *FIB* care transmite secvenţa Fibonacci prin canalul din dreapta

$$FIB \textbf{ sat } (dreapta \leq \langle 1,1\rangle \vee (\langle 1,1\rangle \leq dreapta \wedge dreapta'' \leq dreapta' + dreapta))$$

□

X6 O variabilă cu valoarea x transmite în dreapta cea mai recentă valoare recepţionată la intrare în stânga sau x dacă nu a fost nimic la intrare. Mai formal, dacă cea mai recentă acţiune a fost o transmisie, atunci valoarea care a fost transmisă este egală cu ultimul termen din secvenţa $\langle x\rangle \hat{}\, stânga$

$$VAR_x \textbf{ sat } (canal\,(\overline{ur_0}) = dreapta \Rightarrow \overline{dreapta_0} = \overline{(\langle x\rangle \hat{}\ stînga)_0})$$

unde $\overline{s_0}$ este ultimul element al lui s (paragraful 1.9.5).

Acesta este un exemplu de proces care poate fi specificat adecvat numai în termenii unei secvențe de mesaje pe canale separate proprii. Este de asemenea necesar să știm ordinea comunicațiilor întrețesute pe canalele separate, de exemplu că ultima comunicație este prin canalul din dreapta. În general, această extra complexitate va fi necesară pentru procese care utilizează operatorul alegere. □

X7 Procesul *UNIFICĂ* produce o întrețesere (paragraful 1.9.3) a două secvențe recepționate prin *stânga*1 și *stânga*2, reținând numai cel mult un mesaj

$$UNIFICĂ \textbf{ sat } \exists\, r.dreapta \overset{1}{\leq} r \wedge r\ întrețese(stânga1, stânga2)$$ □

X8 *BUFFER* **sat** *dreapta*≤*stânga* □

Un proces care satisface specificația (*dreapta*≤*stânga*) descrie comportarea unui protocol de comunicație transparent care garantează livrarea prin dreapta numai a acelor mesaje care au sosit în stânga, în aceeași ordine. Un protocol realizează aceasta în ciuda faptului că locul unde se generează mesajele este complet separat de locul unde sunt ele recepționate și de faptul că mediul de comunicație care interconectează cele două contexte este oarecum nesigur. Exemple vor fi date în paragraful 4.4.5.

4.3 Comunicații

Fie P și Q procese și fie c un canal folosit la transmisie de P și la recepție de Q. Astfel mulțimea conținând toate evenimentele de comunicații de forma $c.v$ este conținută în intersecția alfabetului lui P cu alfabetul lui Q. Când aceste procese se compun concurent în sistemul $(P\|Q)$ poate să apară o comunicație $c.v$ numai când ambele procese se angajează simultan în acel eveniment, cu alte cuvinte când P transmite o valoare v pe canalul c și Q simultan primește aceeași valoare. Un proces receptor este pregătit să accepte *orice* valoare comunicabilă astfel că este sarcina procesului emițător de a determina ce valoare–mesaj este transmisă cu fiecare ocazie, ca în 2.6 X4.

Transmisia poate fi astfel privită ca un caz special al operatorului prefix și recepția ca un caz special al alegerii, aceasta conducându-ne la legea

L1 $(c!v \to P) \| (c?x \to Q(x)) = c!v \to (P \| Q(v))$

De remarcat că $c!\nu$ rămâne în dreapta ecuaţiei ca o acţiune observabilă în comportarea sistemului. Aceasta reprezintă posibilitatea fizică de a intercepta firele ce conectează componentele sistemului şi de aceea de a fi la curent cu comunicaţia lor internă. Este de asemenea de ajutor în raţionamentele despre sistem. Dacă se doreşte, astfel de operaţii interne pot fi eliminate prin operatorul de mascare descris în paragraful 3.5 aplicat în afara compunerii paralele a două procese care comunică pe acelaşi canal, cum se arată în legea

L2 $(c!\nu \rightarrow P) \| (c?x \rightarrow Q(x)) \backslash C = (P \| Q(\nu)) \backslash C$

unde $C = \{c.\nu \mid \nu \in \alpha c\}$

Exemple vor fi date în paragraful 4.4 şi 4.5.

Specificarea compunerii paralele a proceselor comunicante ia o formă particulară simplă când numele canalelor sunt utilizate pentru a semnifica secvenţele de mesaje ce trec de-a lungul lor. Fie c numele unui canal prin care P şi Q comunică. În specificarea lui P, c semnifică secvenţa de mesaje comunicate de P prin c. Similar, în specificarea lui Q, c reprezintă secvenţa de mesaje comunicate de Q. Din fericire, prin natura însăşi a comunicaţiei, când P şi Q comunică prin c, secvenţele de mesaje transmise şi recepţionate trebuie tot timpul să fie identice. Deci, această secvenţă trebuie să satisfacă atât specificarea lui P cât şi a lui Q. Aceasta este valabil pentru toate canalele aparţinând intersecţiei alfabetelor lor.

Să considerăm acum un canal din alfabetul lui P dar *nu* şi din cel al lui Q. Acest canal nu poate fi menţionat în specificarea lui Q astfel că valorile transmise pe el sunt constrânse numai de specificarea lui P. Similar, Q este cel care determină proprietăţile comunicaţiei pe propriile sale canale. Prin urmare, o specificare a comportării lui $(P \| Q)$ poate fi simplu formată printr-o conjuncţie logică între specificările lui P şi a lui Q. Totuşi, această simplificare este valabilă numai dacă specificările lui P şi ale lui Q sunt exprimate în întregime în termenii numelor canalelor, care nu este totdeauna posibil aşa cum se arată în 4.2.2 X6.

Exemple

X1 Fie $P = (stânga?x \rightarrow mij!(x \times x) \rightarrow P)$

$Q = (mij?y \rightarrow dreapta!(173 \times y) \rightarrow Q)$

De fapt,

P **sat** $(mij \overset{1}{\leq} pătrat^*(stânga))$

şi Q **sat** $(dreapta \overset{1}{\leq} 173 \times mij)$

unde $(173 \times mij)$ multiplică fiecare mesaj al lui *mij* cu 173.

Urmează că

$$(P \| Q) \textbf{ sat } (dreapta \stackrel{1}{\leq} 173\ mij) \wedge (mij \stackrel{1}{\leq} pătrat^{*}(stânga))$$

Aici specificarea implică

$$dreapta \leq 173 \times pătrat^{*}(stânga)$$

care a fost de fapt intenția originală. □

Când procesele comunicante sunt conectate prin operatorul concurență ||, formulele ce rezultă sunt foarte sugestive pentru o implementare fizică în care componentele electronice sunt conectate prin fire prin care să comunice. Scopul unei astfel de implementări este de a mări viteza de producere a rezultatelor utile. Tehnica este mai ales efectivă când același calcul poate să fie realizat pe fiecare membru al unui flux de date de intrare și rezultatele trebuie să fie transmise cu aceeași viteză ca și recepția, posibil după o întârziere inițială. Astfel de sisteme se numesc *rețele de fluxuri de date* (*data flow networks*).

O reprezentare a unui sistem de procese comunicante se apropie mult de realizare lor fizică. Un canal de transmisie a unui proces este legat la un canal de recepție cu același nume al altui proces, lăsându-se libere doar canalele proprii din alfabetele proceselor. Astfel exemplul X1 poate fi desenat ca în fig 4.2.

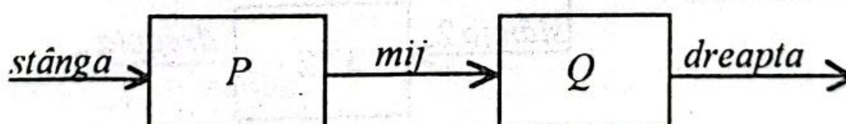


Figura 4.2

X2 Două fluxuri de numere trebuie recepționate prin canalele *stânga*1 și *stânga*2. Pentru fiecare x recepționat prin *stânga*1 și fiecare y prin *stânga*2 numărul $(a \times x + b \times y)$ trebuie transmis prin dreapta. Necesitățile de viteză impun ca multiplicările să se facă concurent. De aceea definim două procese și le compunem

$$X21 = (stânga1?x \rightarrow mij!(a \times x) \rightarrow X21)$$
$$X22 = (stânga2?y \rightarrow mij?z \rightarrow dreapta!(z + b \times y) \rightarrow X22)$$
$$X2 = (X21 \| X22)$$

De fapt, $X2 \textbf{ sat } (mij \stackrel{1}{\leq} a \times stânga1 \wedge dreapta \stackrel{1}{\leq} mij + b \times stânga2)$
$\Rightarrow (dreapta \leq a \times stânga1 + b \times stânga2)$ □

X3 Un flux de numere este recepţionat prin stânga iar în dreapta este transmisă o sumă ponderată a perechilor consecutive de numere introduse, cu ponderile *a* şi *b*.
Mai precis, vrem ca

$$dreapta \le a \times stânga + b \times stânga'$$

Soluţia se poate contura prin adăugarea unui nou proces X23 la soluţia lui X2

$$X3 = (X2 \| X23)$$

unde $X23$ **sat** $(stânga1 \stackrel{1}{\le} stânga \wedge stânga2 \stackrel{1}{\le} stânga')$

X23 poate fi definit

$$X23 = (stânga?x \to stânga1!x \to (\mu X.stânga?x \to stânga2!x \to stânga1!x \to X))$$

Practic procesul copie valoarea recepţionată prin *stânga* atât la *stânga*1 cât şi la *stânga*2, dar omite primul element în cazul lui *stânga*2.

O imagine a reţelei *X*3 este în fig. 4.3. □

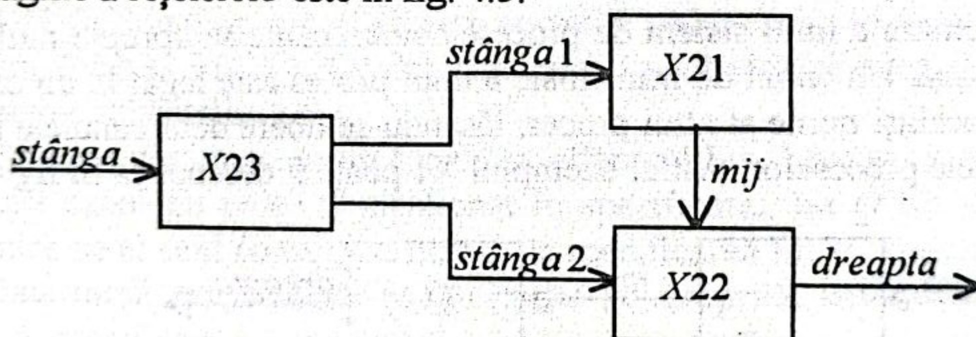


Figura 4.3

Când două procese concurente comunică între ele prin transmitere şi recepţionare numai pe un singur canal, ele nu se pot bloca (compară 2.7 L2). Rezultă că orice reţea de procese ce funcţionează continuu şi este fără cicluri nu se poate bloca deoarece un graf fără cicluri poate fi descompus în subgrafuri componente conectate numai printr-un singur arc. Totuşi reţeaua din *X*3 conţine un ciclu indirect iar reţelele ciclice nu pot fi descompuse în subreţele cu excepţia conexiunilor pe două sau mai multe canale. Astfel, în acest caz, absenţa blocajului nu poate fi atât de uşor asigurată. De exemplu, dacă două ieşiri $stânga2!x \to stânga1!x \to$ din bucla lui *X*23 sunt inversate, blocajul ar putea apare imediat. Pentru a dovedi absenţa blocajului este posibil de multe ori să ignorăm conţinutul mesajelor şi să privim fiecare comunicaţie pe canalul *c* ca un singur eveniment numit *c*. Comunicaţiile pe canalele neconectate pot fi ignorate. Astfel *X*3 poate fi scris în termenii acestor evenimente

$(\mu X.stânga1 \rightarrow mij \rightarrow X)$
$\|(\mu Y.stânga2 \rightarrow mij \rightarrow Y)$
$\|(stânga1 \rightarrow (\mu Z.stânga2 \rightarrow stânga1 \rightarrow Z))$
$= \mu X3.(stânga1 \rightarrow stânga2 \rightarrow mij \rightarrow X3)$

Folosind metode algebrice ca în 2.3 X1, cele de mai sus dovedesc că $X3$ nu se poate bloca.

Aceste exemple ne arată cum pot fi construite rețele de fluxuri de date pentru a calcula unul sau mai multe fluxuri de rezultate din unul sau mai multe fluxuri de date de intrare. Forma rețelei corespunde îndeaproape structurii operanzilor și operatorilor ce apar în expresiile de calcul. Când aceste modele sunt mari, dar regulate, este convenabil să introducem nume subscrise pentru canale și notație iterativă pentru combinația concurentă

$$\|_{i<n} P(i) = (P(0)\|P(1)\|...\|P(n-1))$$

O rețea regulată de acest fel este cunoscută ca *masiv iterativ*. Dacă diagrama de conexiune nu are cicluri directe, se utilizează adesea termenul de *masiv sistolic*, deoarece datele trec prin sistem la fel ca sângele prin camerele inimii.

X4 Canalele $\{stânga_j \mid j<n\}$ sunt folosite pentru a prelua coordonatele unor puncte succesive într-un spațiu n-dimensional. Fiecare mulțime de coordonate trebuie multiplicată cu un vector fix V de lungime n și produsul scalar rezultat trebuie transmis în dreapta. Mai formal

$$dreapta \leq \sum_{j=0}^{n-1} V_j \times stânga_j$$

Se specifică că în fiecare microsecundă cele n coordonate ale unui punct trebuie recepționate și produsul scalar transmis. Viteza fiecărui procesor individual este astfel că îi trebuie aproape 1 μs pentru o recepție, o multiplicare, o adunare și o transmisie. De aceea, este clar că este nevoie de cel puțin n procesoare care să lucreze concurent. Soluția problemei ar putea fi realizată printr-un masiv iterativ cu cel puțin n elemente.

Să înlocuim în specificație operatorul Σ cu definiția lui inductivă

$mij_0 = 0^*$
$mij_{j+1} = V_j \times stânga_j + mij_j,$ pentru $j<n$
$dreapta = mij_n$

Astfel, am despărţit specificaţia într-o conjuncţie de $n+1$ ecuaţii componente, fiecare conţinând cel mult o multiplicare. Tot ce se cere este de a scrie un proces pentru fiecare ecuaţie

$$MULT_0=(\mu X.mij_0!0\to X)$$
$$MULT_{j+1}=(\mu X.stânga_j?x\to mij_j?y\to mij_{j+1}!(V_j\times x+y)\to X) \quad \text{pentru } j<n$$
$$MULT_{n+1}=(\mu X.mij_n?x\to dreapta!x\to X)$$
$$REŢEA=\|_{j<n+2} MULT_j$$

Diagrama de conexiune este dată în fig. 4.4. □

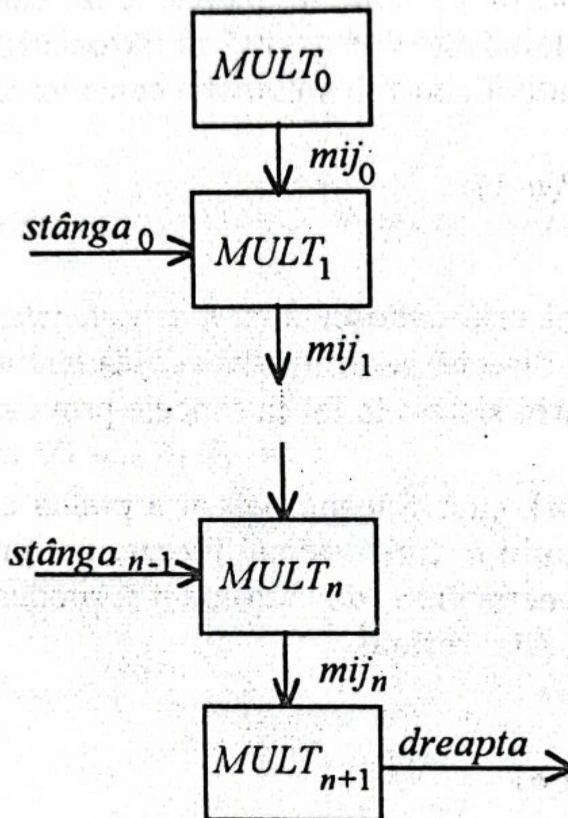


Figura 4.4

X5 Acest exemplu este similar cu X4 cu excepţia faptului că cele m produse scalare diferite ale aceleiaşi mulţimi de coordonate sunt necesare aproape simultan. Practic, canalul $stânga_j$ (pentru $j<n$) trebuie folosit pentru recepţia coloanei j a unui masiv infinit, aceasta la rândul ei trebuie multiplicată cu matricea $M(n\times m)$ şi coloana i a rezultatului trebuie transmisă prin $dreapta_i$ pentru $i<m$. Ca formulă

$$dreapta_i=\sum_{j<n} M_{ij}\times stânga_j$$

Coordonatele rezultatului sunt necesare mai rapid ca înainte astfel că sunt necesare cel puţin $m\times n$ procese.

Soluția poate găsi aplicație practică pentru un dispozitiv de afișare grafică care transformă automat sau chiar rotește o reprezentare bidimensională a unui obiect tridimensional. Forma este definită de o serie de puncte într-un spațiu absolut. Masivul iterativ aplică transformările liniare necesare pentru a calcula deflecția pe plăcile x și y ale unui tub cu raze catodice. O a treia coordonată de ieșire ar putea controla intensitatea spotului.

Soluția se bazează pe cele reprezentate în fig. 4.5. Fiecare coloană a masivului (cu excepția ultimei) este modelată de soluția din X4. Astfel se copie către vecinul său prin canalul de transmisie orizontal fiecare valoare recepționată prin canalul de intrare orizontal .

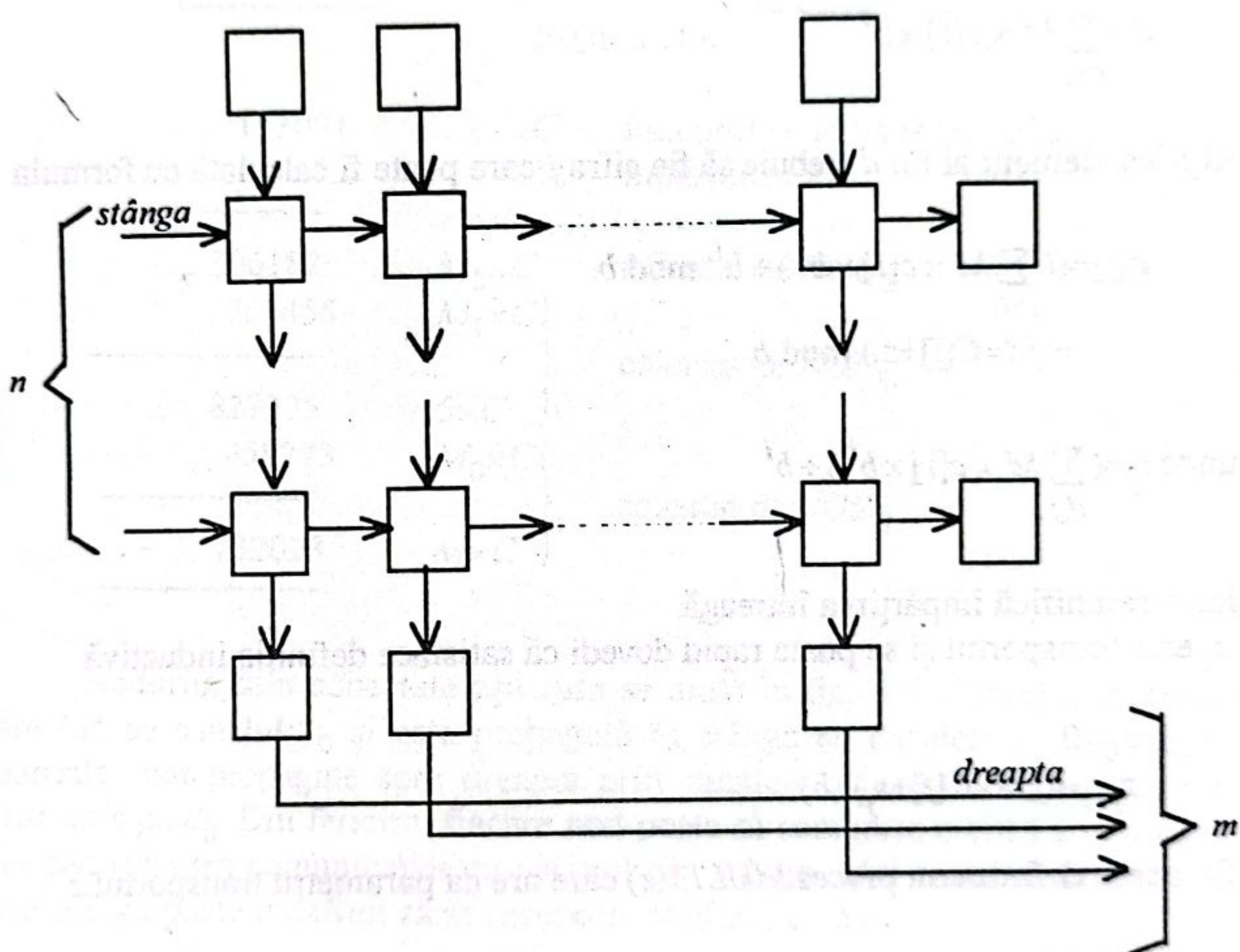


Figura 4.5

Procesele din marginea dreaptă pur și simplu renunță la valorile pe care le recepționează. Astfel, ar fi posibilă o economie de procesoare prin înglobarea funcțiilor celor marginale în acelea ale vecinilor.

Detaliile soluției sunt lăsate ca exercițiu. □

X6 Recepția pe canalul c poate fi interpretată ca succesiunea cifrelor unui număr natural C pornind de la cifra cea mai puțin semnificativă și exprimate în baza b. Definim valoarea numărului recepționat ca

$$C=\sum_{i\geq 0} c[i] \times b^i$$

unde $c[i]<b$ pentru toţi i

Dat un multiplicator fix M (o cifră), transmisia pe canalul d este o succesiune de cifre ale produsului $M\times C$. Cifrele trebuie transmise cu o întârziere minimă. Să specificăm problema mai precis. Valoarea dorită a transmisiei d este

$$d=\sum_{i\geq 0} M \times c[i] \times b^i$$

Al j-lea element al lui d trebuie să fie cifra j care poate fi calculată cu formula

$$d[j]=((\sum_{i\geq 0} M \times c[i] \times b^i) \div b^i \textbf{ mod } b$$
$$=(M\times C[j]+z_j) \textbf{ mod } b$$

unde $z_j=(\sum_{i<j} M \times c[i] \times b^i) \div b^i$

iar $\div$ semnifică împărţirea întreagă.
z_j este transportul şi se poate rapid dovedi că satisface definiţia inductivă

$$z_0=0$$
$$z_{j+1}=((M\times C[j]+z_j)\div b)$$

De aceea definim un proces $MULT1(z)$ care are ca parametru transportul z

$$MULT1(z)=c?x\rightarrow d!(M\times x+z) \textbf{ mod } b\rightarrow MULT1((M\times x+z)\div b)$$

Valoarea iniţială pentru z este 0 astfel că soluţia cerută este

$$MULT=MULT1(0)$$ □

X7 Problema este aceeaşi ca în X6 cu deosebirea că M este un număr cu mai multe cifre

$$M=\sum_{i<n} M_i \times b^i$$

Am văzut cum un singur procesor poate multiplica numere doar cu o singură cifră. De altfel şi transmisia se produce cu o viteză care permite o multiplicare numai pe cifră. Prin urmare, cel puţin n procesoare sunt necesare. Vom face ca fiecare proces NOD_i să gestioneze o cifră M_i a multiplicatorului.

Baza soluţiei este tradiţionalul algoritm manual pentru înmulţire cu mai multe cifre, cu deosebirea că sumele parţiale sunt adunate imediat la următorul rând al tabelei.

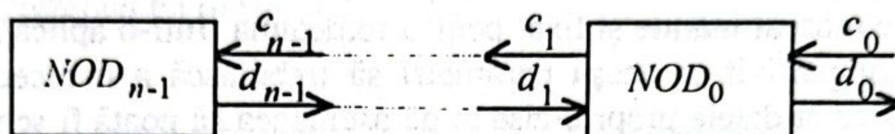


Figura 4.6

..... 153091	C	numărul ce se va recepţiona
253	M	multiplicatorul
.... 306182	$M_2 \times C$	calculat de NOD_2
.... 765455	$M_1 \times C$	calculat de NOD_1
..... 827275	$5 \times C$	
... 459273	$M_0 \times C$	calculat de NOD_0
... 732023	$M \times C$	

Nodurile sunt conectate aşa cum se arată în fig. 4.6. Recepţia originală are loc pe canalul c_0 şi este propagată în stânga pe canalele c. Rezultatele parţiale sunt propagate spre dreapta prin canalele d iar rezultatul final este transmis pe d_0. Din fericire, fiecare nod poate să comunice o cifră a rezultatului său înaintea comunicaţiei cu vecinul din stânga. Mai mult, nodul cel mai din stânga poate fi definit să se comporte conform cu X6

$$NODE_{n-1}(z) = c_{n-1}?x \rightarrow d_{n-1}!(M_{n-1} \times x + z) \textbf{ mod } b$$
$$\rightarrow NODE_{n-1}((M_{n-1} \times x + z) \div b)$$

Nodurile rămase sunt la fel cu celelalte cu deosebirea că fiecare din ele pasează cifra recepţionată vecinului său stâng şi adună ce primeşte de la acelaşi vecin stâng la propriul transport. Pentru $k<n-1$

$$NODE_k(z) = c_k?x \rightarrow d_k!(M_k \times x + z) \textbf{ mod } b \rightarrow c_{k+1}!x \rightarrow d_{k+1}?y$$
$$\rightarrow NODE_k(y + (M_k \times x + z) \div b)$$

Întreaga reţea este definită

$$\underset{i<n}{\|} NODE_i(0)$$ □

X7 este un exemplu simplu dintr-o clasă de algoritmi ingenioşi pentru reţele în care există un ciclu principal în graful orientat al canalelor de comunicaţie. Esenţa problemei a fost mult simplificată de presupunerea că multiplicatorul este cunoscut înainte şi fixat pentru totdeauna. Într-o aplicaţie practică este mult mai potrivit ca aceşti parametri să trebuiască a fi recepţionaţi pe acelaşi canal ca şi datele propriu-zise şi de asemenea să poată fi schimbaţi ori de câte ori e necesar. Desigur, această implementare necesită mult mai mare grijă dar şi puţină ingeniozitate.

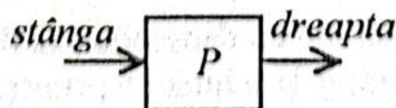
O metodă simplă de implementare este de a introduce un simbol special, de exemplu *reîncărcare* ca un indiciu că următorul număr sau numere trebuie tratate ca o schimbare de parametru, iar dacă numărul parametrilor este variabil trebuie de asemenea introdus simbolul *sfârşit_reîncărcare*.

X8 În acest exemplu avem aceeaşi situaţie ca în X4 cu excepţia faptului că parametrii V_j trebuie reîncărcaţi cu numărul ce urmează imediat unui simbol *reîncărcare*. Definiţia lui $MULT_{j+1}$ trebuie schimbată pentru a include multiplicatorul ca parametru

$MULT_{j+1}$=*stânga*$_j$?x→
  if x=*reîncărcare* **then** (*stânga*$_j$?y→$MULT_{j+1}$ (y))
    else (mij_j?y→mij_{j+1}!(v×x+y)→$MULT_{j+1}$(v)) □

4.4 Conducte

În acest paragraf ne vom concentra atenţia la procesele cu numai două canale în alfabet, numite canal de recepţie (intrare) *stânga* şi canal de transmisie (ieşire) *dreapta*. Astfel de procese se numesc *conducte* şi pot fi reprezentate ca în fig. 4.7.



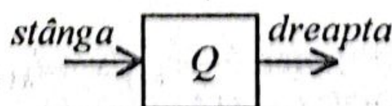


Figura 4.7

Procesele P și Q pot fi legate astfel încât canalul *dreapta* al lui P să fie conectat la canalul *stânga* al lui Q și secvența mesajelor transmise de P și recepționate de Q pe acest canal intern să fie mascat mediului lor. Rezultatul acestei conexiuni este notat

$P \gg Q$

și poate fi reprezentat ca în fig. 4.8.

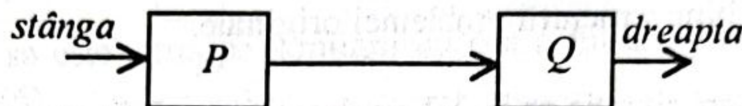


Figura 4.8

Această diagramă de conexiune ne arată mascarea canalului de conexiune prin absența unui nume. De asemenea desenul arată că toate mesajele recepționate de canalul *stânga* al lui $(P \gg Q)$ sunt recepționate de P și toate mesajele emise pe canalul din dreapta a lui $(P \gg Q)$ sunt emise de Q. În final $(P \gg Q)$ este el însuși o conductă și poate fi legat în serie cu alte conducte

$(P \gg Q) \gg R$, $(P \gg Q) \gg (R \gg S)$ etc.

Din 4.4.1 L1 operatorul $\gg$ este asociativ, astfel că pe viitor vom omite parantezele într-o astfel de succesiune.

Validitatea înlănțuirii proceselor prin $\gg$ depinde evident de constrângerile de alfabet

$\alpha(P \gg Q) = \alpha stânga(P) \cup \alpha dreapta(Q)$

iar o altă constrângere afirmă despre canalele conectate că sunt capabile să transmită numai aceleași mesaje

$\alpha dreapta(P) = \alpha stânga(Q)$

Exemple

X1 O conductă care transmite fiecare valoare recepționată multiplicată cu patru (4.2 X2)

$4ORI = DUBLU \gg DUBLU$



X2 Un proces care recepţionează cartele de 80 caractere şi transmite conţinutul lor împachetat în linii de câte 125 de caractere (4.2 X3, X4)

DESPACHET≫*ÎMPACHET*

Acest proces este cam dificil de scris utilizând tehnicile de programare structurată convenţionale deoarece nu este clar dacă bucla principală este bine să se facă pentru recepţia fiecărei cartele sau pentru transmisia fiecărei linii. Problema este cunoscută de Michael Jackson ca *dezacordul de structură*. Soluţia dată mai sus conţine o buclă separată în fiecare din cele două procese, care se potriveşte foarte bine structurii problemei originale. □

X3 Avem aceeaşi situaţie ca în X2 cu deosebirea că fiecare pereche de asteriscuri consecutive este înlocuită cu "↑" (4.2 X5)

DESPACHET≫*ELIMIN*≫*ÎMPACHET*

Într-un program secvenţial convenţional această schimbare minoră în specificare ar putea cauza probleme severe. Este uşor de evitat astfel de probleme prin simpla inserare a unui nou proces. Acest fel de modularitate a fost introdus şi exploatat de proiectanţii de sisteme de operare. □

X4 Aceeaşi situaţie ca în X2 cu excepţia faptului că cititul cartelelor poate continua când imprimanta este neoperaţională, iar imprimatul poate continua când cititorul este neoperaţional (4.2 X9)

DESPACHET≫*BUFFER*≫*ÎMPACHET*

Bufferul ţine caracterele care au fost produse de procesul *DESPACHET* dar n-au fost încă consumate de procesul *ÎMPACHET*. Ele vor fi disponibile pentru recepţia lui *ÎMPACHET*, timp în care procesul *DESPACHET* este temporar întârziat. Bufferul netezeşte astfel variaţiile temporare în viteza de producţie şi consum. Totuşi soluţia nu poate rezolva problema nepotrivirii pe termen lung a vitezelor de producţie şi consum. Dacă cititorul de cartele este în medie mai lent decât imprimanta, bufferul va fi aproape tot timpul gol şi nu se va realiza nici un efect de potrivire. Dacă cititorul este mai rapid, bufferul se va extinde la infinit până când consumă tot spaţiul disponibil de stocare. □

X5 Pentru a evita expansiunea nedorită a bufferelor, se obişnuieşte să se limiteze numărul mesajelor stocate. Chiar şi un singur buffer realizat de procesul *COPIE* (4.2 X1) poate fi adecvat. Iată o versiune a lui X4 care citeşte în avans o cartelă la intrare şi stochează o linie de imprimat la ieşire

COPIE≫DESPACHET≫ÎMPACHET≫COPIE

De notat că alfabetele celor 2 procese *COPIE* sunt diferite, lucru care trebuie înțeles din contextul în care sunt plasate. □

X6 Un buffer dublu care acceptă până la două mesaje înainte de a se face transmisia primului

COPIE≫COPIE □

Comportarea sa este aproape similară cu cea a lui *LANȚ2* (2.6 X4) și chiar *AVS2* (1.1.3 X6).

4.4.1 Legi

Cea mai utilă proprietate algebrică a înlănțuirii este asociativitatea

L1 *P≫(Q≫R)=(P≫Q)≫R*

Restul legilor prezintă cum recepția și transmisia pot fi implementate printr-o conductă. Ele permit ca descrierile proceselor să fie simplificate la o formă de execuție simbolică. De exemplu, dacă procesul din stânga lui ≫ începe cu transmisia unui mesaj ν prin canalul *dreapta* și procesul din dreapta lui ≫ începe cu recepția de la *stânga*, mesajul ν este transmis de la primul proces la cel de-al doilea. Totuși comunicația ce are loc este mascată, așa cum se arată în legea

L2 *(dreapta!ν→P)≫(stânga?y→Q(y))=P≫Q(ν)*

Dacă unul din procese este determinat să comunice cu celălalt, dar celălalt este pregătit să comunice extern, are loc mai întâi comunicația externă iar comunicația internă este amânată până la o ocazie ulterioară

L3 *(dreapta!ν→P)≫(dreapta!w→Q)=dreapta!w→((dreapta!ν→P)≫Q)*

L4 *(stânga?x→P(x))≫(stânga?y→Q(y))*
=stânga?x→(P(x)≫(stânga?y→Q(y)))

Dacă ambele procese sunt pregătite pentru comunicație externă, oricare din ele poate avea loc mai întâi

L5 *(stânga?x→P(x))≫(dreapta!w→Q)=(stânga?x→(P(x)≫(dreapta!w→Q))*
|dreapta!w→((stânga?x→P(x))≫Q))

Legea L5 este de asemenea valabilă când operatorul $\gg$ este înlocuit cu $\gg R \gg$, deoarece conductele din interiorul unui lanţ nu pot comunica direct cu mediul

L6 $(\textit{stânga}?x \to P(x)) \gg R \gg (\textit{dreapta}!w \to Q) =$
$(\textit{stânga}?x \to (P(x) \gg R \gg (\textit{dreapta}!w \to Q)) \mid \textit{dreapta}!w \to ((\textit{stânga}?x \to P(x)) \gg R \gg Q))$

Generalizări analoage pot fi făcute altor legi.

L7 Dacă R este un lanţ de procese, toate pornind cu transmisia prin dreapta

$$R \gg (\textit{dreapta}!w \to Q) = \textit{dreapta}!w \to (R \gg Q)$$

L8 Dacă R este un lanţ de procese, toate pornind cu recepţie din stânga

$$(\textit{stânga}?x \to P(x) \gg R = \textit{stânga}?x \to (P(x) \gg R)$$

Exemple

X1 Să definim

$$R(y) = (\textit{dreapta}!y \to COPIE) \gg COPIE$$

Astfel $R(y) = (\textit{dreapta}!y \to COPIE) \gg (\textit{stânga}?x \to \textit{dreapta}!x \to COPIE)$ — definiţia lui *COPIE*

$= COPIE \gg (\textit{dreapta}!y \to COPIE)$ — L2 □

X2 $COPIE \gg COPIE$

$= (\textit{stânga}?x \to \textit{dreapta}!x \to COPIE) \gg COPIE$ — definiţia lui *COPIE*

$= (\textit{stânga}?x \to ((\textit{dreapta}!x \to COPIE) \gg COPIE)$ — L4

$= \textit{stânga}?x \to R(x)$ — definiţia lui $R(x)$ □

X3 Din ultima linie a lui X1 deducem

$R(y) = (\textit{stânga}?x \to \textit{dreapta}!x \to COPIE) \gg (\textit{dreapta}!y \to COPIE)$

$= (\textit{stânga}?x \to ((\textit{dreapta}!x \to COPIE) \gg (\textit{dreapta}!y \to COPIE))$
$\mid \textit{dreapta}!y \to (COPIE \gg COPIE))$ — L5

$= (\textit{stânga}?x \to \textit{dreapta}!y \to R(x)$ — L3
$\mid \textit{dreapta}!y \to \textit{stânga}?x \to R(x))$ — X2

Aceasta ne arată că un buffer dublu, după primirea primului mesaj este pregătit atât să transmită acel mesaj cât şi să recepţioneze un al doilea înainte

de a face transmisia. Raţionamentele pentru cele de mai sus sunt foarte asemănătoare cu cele din 2.3.1 X1. □

4.4.2 Implementare

În implementarea lui $P \gg Q$ se disting trei cazuri

(1) Dacă comunicaţia poate avea loc pe canalul de conexiune intern, aceasta se produce imediat, fără considerarea mediului extern. Dacă este posibil o secvenţă infinită de astfel de comunicaţii, procesul diverge (paragraful 3.5.2.).

(2) Altfel, dacă mediul este interesat în comunicaţia pe canalul stâng, aceasta este tratată de *P*.

(3) Dacă mediul este interesat de comunicaţia prin canalul drept, aceasta este tratată de *Q*.

Pentru explicitarea operaţiilor de recepţie şi transmisie, vezi paragraful 4.2.1

```
lanţ(P,Q)=
if P("dreapta)≠"BLIP∧Q ("stânga)≠"BLIP
  then lanţ(cdr(P("dreapta)), Q("stânga)(car(P("dreapta))))      caz (1)
else λx. if x="dreapta
     then if Q("dreapta)="BLIP then "BLIP
          else cons(car(Q("dreapta)),
                    lanţ(P, cdr(Q("dreapta))))                    caz (3)
     else if x="stânga
          then if P(x)="BLIP then "BLIP
                    else λy.lanţ(P("stânga) (y),Q)                caz (2)
     else "BLIP
```

4.4.3 Blocaj prin ciclare infinită

Operatorul de înlănţuire conectează două procese printr-un singur canal, de aceea nu există riscul de blocaj. Dacă atât *P* cât şi *Q* sunt bucle, atunci nici $(P \gg Q)$ nu se va opri. Din nefericire, există un nou pericol şi anume acela ca procesele *P* şi *Q* să-şi petreacă tot timpul comunicând între ele, astfel încât $(P \gg Q)$ să nu mai comunice cu lumea externă. Acest caz de divergenţă (paragraful 3.5.1, 3.8) este ilustrat de exemplul banal

$P = (dreapta!1 \rightarrow P)$
$Q = (stânga!x \rightarrow Q)$

$(P \gg Q)$ este evident un proces inutil. El este chiar mai rău decât *STOP* prin faptul că o buclă fără sfârşit poate consuma resurse de calcul fără limită şi fără a produce ceva. Un exemplu mai puţin banal este $(P \gg Q)$ unde

$$P = (dreapta!1 \rightarrow P \mid stânga?x \rightarrow P1(x))$$
$$Q = (stânga?x \rightarrow Q \mid dreapta!1 \rightarrow Q1)$$

În acest exemplu, divergenţa provine din simpla posibilitate de comunicare internă infinită. Divergenţa există chiar dacă este oferită alternativa unei comunicări externe prin canalele *stânga* şi *dreapta* cu orice ocazie posibilă şi chiar dacă după o astfel de comunicare externă, următoarea comportare a lui $(P \gg Q)$ n-ar diverge.

O metodă simplă de a demonstra că $(P \gg Q)$ este lipsit de blocajul prin ciclare infinită este de a arăta că P este cu *gardă-stângă* în sensul că nu poate transmite niciodată o serie infinită de mesaje prin *dreapta* fără a se interpune recepţii prin *stânga*. Pentru a ne asigura de aceasta, trebuie să dovedim că lungimea secvenţei transmise prin *dreapta* este oricând mărginită superior de o funcţie bine definită f a secvenţei de valori recepţionate prin *stânga*, sau mai formal definim

$$P \text{ este cu gardă-stângă} \equiv \exists f . P \textbf{ sat } (\#dreapta \leq f(stânga))$$

Proprietatea de gardă-stângă este adesea evidentă din scrierea lui P.

L1 Dacă orice recursivitate utilizată în definirea lui P este cu gardă datorită recepţiei prin stânga, atunci P este cu gardă-stângă.

L2 Dacă P este cu gardă-stânga, atunci $(P \gg Q)$ este liber de blocaj prin ciclare infinită.

Exact acelaşi raţionament se aplică pentru a pune în evidenţă garda-dreaptă a celui de-al doilea operand al lui $\gg$.

L3 Dacă Q este cu gardă-dreaptă, atunci $(P \gg Q)$ este liberă de blocaj prin ciclare infinită.

Exemple

X1 Următoarele procese sunt cu gardă-stângă datorită lui L1 (4.1 X1, X2, X5, X9)

COPIE, *DUBLU*, *ELIMIN*, *BUFFER* □

X2 Următoarele procese sunt cu gardă-stângă potrivit definiției originale, din cauză că

DESPACHET **sat** $\#dreapta \leq \#(\,{}^\frown/stânga)$
ÎMPACHET **sat** $\#dreapta \leq \#stânga$ □

X3 *BUFFER* nu este cu gardă-dreaptă deoarece poate recepționa arbitrar de multe mesaje în stânga fără să le transmită vreodată în dreapta. □

4.4.4 Specificații

Specificarea unei conducte poate fi exprimată adesea ca o expresie *S*(*stânga*, *dreapta*) corespunzătoare secvenței de mesaje recepționate pe canalul din stânga și secvenței de mesaje transmise prin dreapta. Când două conducte sunt conectate în serie, secvența *dreapta* produsă de operandul stâng este egalată de secvența *stânga* consumată de operandul drept. Această secvență comună este apoi mascată. Tot ceea ce se știe despre secvența mascată este că ea există. Dar de asemenea vrem să prevenim riscul blocării prin ciclare infinită. De aceea prezentăm regula

L1 Dacă *P* **sat** *S*(*stânga*,*dreapta*)
și *Q* **sat** *T*(*stânga*,*dreapta*)
și dacă *P* este cu gardă-stângă sau *Q* cu gardă-dreaptă
atunci $(P \gg Q)$ **sat** $\exists\, s. S(stânga, s) \wedge T(s, dreapta)$

Această regula afirmă că relația între *stânga* și *dreapta* promovată de $(P \gg Q)$ este o compunere relațională normală între relația lui *P* cu relația lui *Q*. Deoarece operatorul $\gg$ nu poate introduce blocaj în conducte ne putem permite să omitem raționamentul pentru refuzuri.

Exemple

X1 *DUBLU* **sat** $dreapta \overset{1}{\leq} dublu^*(stânga)$
DUBLU este cu gardă-stângă și cu gardă-dreaptă

Astfel $(DUBLU \gg DUBLU)$ **sat** $\exists s.(s \overset{1}{\leq} dublu^*(stânga) \wedge dreapta \overset{1}{\leq} dublu^*(s))$

$\equiv dreapta \overset{2}{\leq} dublu^*(dublu^*(stânga))$

$\equiv dreapta \overset{2}{\leq} 4ori^*(stânga)$ □

X2 Să aplicăm recursivitatea împreună cu $\gg$ pentru a da o altă definiție unui buffer

$BUFF = \mu X.(stânga?x \rightarrow (X \gg (dreapta!x \rightarrow COPIE)))$

Dorim să dovedim că

$BUFF$ **sat** $(dreapta \leq stânga)$

Presupunem că

X **sat** $\# stânga \geq n \vee dreapta \leq stânga$

Ştim că

$COPIE$ **sat** $dreapta \leq stânga$

Rezultă că $(dreapta!x \rightarrow COPIE)$ **sat** $((dreapta = stânga = \langle\rangle$
$\vee (dreapta \geq \langle x \rangle \wedge dreapta' \leq stânga)) \Rightarrow dreapta \leq \langle x \rangle \hat{} stânga$

Deoarece operandul drept este cu gardă-dreaptă, din L1 şi presupunând că

$(X \gg (dreapta!x \rightarrow COPIE))$ **sat** $(\exists s.(\# stânga \geq n \vee s \leq stânga)$
$\wedge dreapta \leq \langle x \rangle \hat{} s)$
$\Rightarrow (\# stânga \geq n \vee dreapta \leq \langle x \rangle \hat{} stânga)$

deducem $stânga?x \rightarrow (...)$ **sat** $dreapta = stânga = \langle\rangle$
$\vee (stânga > \langle\rangle \wedge (\# stânga \geq n \vee dreapta \leq \langle stânga_0 \rangle \hat{} stânga'))$
$\Rightarrow \# stânga \geq n+1 \vee dreapta \leq stânga$

Concluzia dorită rezultă din legea pentru procese recursive (3.7.1 L8). Legea mai simplă (1.10.2 L6) nu poate fi folosită din cauză că recursivităţile nu sunt cu gardă evidentă. □

4.4.5 Buffere şi protocoale

Un buffer este un proces care transmite în partea dreaptă exact aceeaşi secvenţă de mesaje pe care a primit-o în stânga, posibil cu o întârziere. Astfel că, dacă nu este gol este totdeauna gata să transmită în dreapta. Mai formal, definim un buffer ca un proces care nu se opreşte niciodată, care este liber de blocaj prin ciclare infinită şi care satisface specificaţia

P **sat** $(dreapta \leq stânga)$
$\wedge$(**if** $dreapta = stânga$ **then** $stânga \notin ref$ **else** $dreapta \notin ref$)

Aici $c \notin ref$ înseamnă că procesul nu poate refuza comunicarea pe canalul c (paragraful 3.7., 3.4). Urmează că toate bufferele sunt cu gardă-stângă.

Exemplu

X1 Următoarele procese sunt buffere

COPIE, (*COPIE*≫*COPIE*), *BUFF*, *BUFFER* □

Bufferele sunt evident utile în stocarea informației care urmează a fi procesată. Dar sunt chiar și mai utile ca specificații pentru comportarea dorită a unui protocol de comunicație care se intenționează să livreze mesaje în aceeași ordine în care au fost primite. Un astfel de protocol constă din două procese, un transmițător T și un receptor R care sunt conectate în serie ($T \gg R$). Dacă un protocol este corect este clar că ($T \gg R$) trebuie să fie un buffer.

În practică, suportul fizic care conectează transmițătorul de receptor este suficient de lung și mesajele care sunt trimise prin el pot fi deteriorate sau pierdute. Astfel, comportarea cablului însuși poate fi modelată de un proces *FIRE*, care se comportă nu tocmai ca un buffer. Este sarcina proiectantului de protocol să se asigure că în ciuda proastei comportări a suportului (cablu), sistemul ca întreg se comportă ca un buffer, adică

($T \gg FIRE \gg R$) este un buffer

Un protocol este de obicei realizat dintr-un număr de straturi (T_1, R_1), (T_2, R_2), ..., (T_n, R_n), fiecare utilizând stratul anterior drept mediul său de comunicare.

$$Tn \gg \ldots \gg (T_2 \gg (T_1 \gg FIRE \gg R_1) \gg R_2) \gg \ldots \gg R_n$$

Desigur, când protocolul este implementat în practică, toți transmițătorii sunt strânși într-unul singur la un capăt și toți receptorii la celălalt, în concordanță cu modificarea asocierii în paranteze

$$(T_n \gg \ldots \gg T_2 \gg T_1) \gg FIRE \gg (R_1 \gg R_2 \gg \ldots \gg R_n)$$

Asociativitatea pentru ≫ garantează că această regrupare nu schimbă comportarea sistemului.

În practică, protocoalele trebuie să fie mult mai complicate ca acesta. Unidirecționalitatea transferului mesajelor nu este adecvată pentru realizarea unei comunicații fiabile pe un cablu nefiabil. De aceea este necesar să adăugăm canale pentru comunicația inversă permițând receptorului să trimită

înapoi semnale de confirmare a bunei recepționări a mesajelor astfel ca mesajele neconfirmate să poată fi retransmise.

Următoarele legi se dovedesc utile în dovedirea corectitudinii protocoalelor. Ele se datoresc lui A. W. Roscoe.

L1 Dacă P și Q sunt buffere,

sunt și $(P \gg Q)$ și $(stânga?x \to (P \gg (dreapta!x \to Q)))$

L2 Dacă $(T \gg R) = (stânga?x \to (T \gg (dreapta!x \to R)))$ atunci $(T \gg R)$ este un buffer.

Următoarea lege este o generalizare a lui L2

L3 Dacă pentru o funcție f și toți z

$$(T(z) \gg R(z)) = (stânga?x \to (T(f(x,z)) \gg (dreapta!x \to R(f(x,z)))))$$

atunci $T(z) \gg R(z)$ este un buffer pentru toți z.

Exemple

X1 Următoarele procese sunt buffere datorită lui L1

$COPIE \gg COPIE$, $BUFFER \gg COPIE$, $COPIE \gg BUFFER$,
$BUFFER \gg BUFFER$ □

X2 S-a arătat în 4.4.1 X1 și X2 că

$$(COPIE \gg COPIE) = (stânga?x \to (COPIE \gg (dreapta!x \to COPIE)))$$

Din L2 rezultă că membrul stâng este un buffer. □

X3 Codarea în fază.
Un codor în fază este un proces T care recepționează un flux de biți și transmite <0,1> pentru fiecare 0 recepționat și <1,0> pentru fiecare 1 recepționat. Un decodor R inversează această translație

$$T = stânga?x \to dreapta!x \to dreapta!(1-x) \to T$$
$$R = stânga?x \to stânga?y \to \textbf{if } y = x \textbf{ then } EROARE \textbf{ else } (dreapta!x \to R)$$

unde procesul $EROARE$ este lăsat nedefinit.

Dorim să dovedim că $T \gg R$ este un buffer, prin L2

$$(T \gg R) = stânga?x \rightarrow ((dreapta!x \rightarrow dreapta!(1-x) \rightarrow T)$$
$$\gg (stânga?x \rightarrow stânga?y \rightarrow \textbf{if } y=x \textbf{ then } EROARE \textbf{ else } (dreapta!x \rightarrow R)))$$
$$= stânga?x \rightarrow (T \gg \textbf{if } (1-x)=x \textbf{ then } EROARE \textbf{ else } (dreapta!x \rightarrow R))$$
$$= stânga?x \rightarrow (T \gg (dreapta!x \rightarrow R))$$

De aceea $(T \gg R)$ este un buffer prin L2. □

X4 Schimbarea unui bit.
Transmițătorul *T* reproduce cu fidelitate biții recepționați prin canalul din stânga spre canalul din dreapta cu excepția cazului când după trei de 1 consecutivi care au fost transmiși, schimbă un bit în 0. Astfel intrarea 01011110 este transmisă ca 010111010. Receptorul *R* trebuie să facă operația inversă. Astfel, trebuie arătat că $(T \gg R)$ este un buffer. Construcția lui *T* și a lui *R* și dovedirea corectitudinii sunt lăsate ca exerciții. □

X5 Partajarea liniei.
Se dorește copierea unor date de la un canal *stânga*1 la *dreapta*1 și de la *stânga*2 la *dreapta*2. Aceasta poate fi cel mai ușor realizat prin două protocoale disjuncte fiecare utilizând alt cablu. Din nefericire, numai un singur cablu *mij* este disponibil și el trebuie folosit de ambele fluxuri de date ca în fig. 4.9.

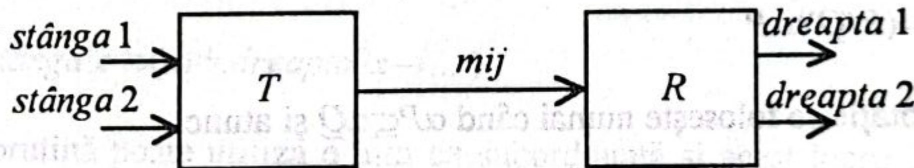


Figura 4.9

Mesajele recepționate de *T* trebuie marcate înaintea transmiterii prin *mij* iar *R* trebuie să le demarcheze și să le transmită pe canalul corespunzător din dreapta.

$$T = (stânga1?x \rightarrow mij!marc1(x) \rightarrow T \mid stânga2?y \rightarrow mij!marc2(y) \rightarrow T)$$
$$R = (mij?z \rightarrow \textbf{if } marc(z)=1 \textbf{ then } (dreapta1!demarc(z) \rightarrow R)$$
$$\textbf{else } (dreapta2?demarc(z) \rightarrow R))$$

Soluția este destul de nesatisfăcătoare. Dacă două mesaje sunt recepționate pe *stânga*1, dar receptorul nu este încă gata pentru ele, întregul sistem va trebui să aștepte iar transmisia între *stânga*2 și *dreapta*2 este serios întârziată. Dacă introducem buffere pe canale, amânăm numai puțin problema. Soluția corectă este de a introduce un alt canal în direcția inversă iar *R* să trimită semnale înapoi lui *T* oprind transmiterea mesajelor pe direcția pentru care pare să fie

cerere puţină. Această problemă este cunoscută drept *controlul fluxului* (*flow control*). □

4.5 Subordonarea

Fie P şi Q procese cu

$$\alpha P \subseteq \alpha Q$$

În combinaţia $(P\|Q)$ fiecare acţiune a lui P poate apare numai când Q îi permite să apară. Cu toate acestea Q se poate angaja independent în acţiunile lui $(\alpha Q - \alpha P)$ fără cunoştinţa şi permisiunea partenerului său P. Astfel P este pentru Q un fel de proces slave sau subordonat, în timp ce Q se comportă ca un proces principal sau master. Când comunicaţiile între un proces subordonat şi unul principal trebuie mascate mediului lor comun folosim notaţia asimetrică

$$P /\!/ Q$$

Utilizând operatorul de mascare, aceasta poate fi definită

$$P /\!/ Q = (P /\!/ Q) \backslash \alpha P$$

Această notaţie se foloseşte numai când $\alpha P \subseteq \alpha Q$ şi atunci

$$\alpha(P /\!/ Q) = (\alpha Q - \alpha P)$$

Se obişnuieşte să i se dea procesului subordonat un nume, fie m, care este folosit în procesul principal pentru toate interacţiunile cu subordonatul. Tehnica de numire a proceselor descrisă în paragraful 2.6.2 poate fi rapid extinsă la procese comunicante prin introducerea numelor de canal compuse. Acestea iau forma $m.c$, unde m este un nume de proces şi c este unul din canalele sale. Fiecare comunicaţie pe acest canal este un triplet

$$m.c.v$$

unde $\alpha m.c(m{:}P) = \alpha c(P)$ şi $v \in \alpha c(P)$

În construcţia $(m{:}P /\!/ Q)$, Q comunică cu P de-a lungul canalelor cu nume compuse de forma $m.c$ şi $m.d$. Avem în vedere de asemenea că P foloseşte canalele corespunzătoare simple c şi d pentru aceleaşi comunicaţii. Astfel, de exemplu

$(m{:}(c!v \rightarrow P) /\!/ (m.c?x \rightarrow Q(x))) = (m{:}P /\!/ Q(v))$

Deoarece toate aceste comunicaţii sunt mascate mediului numele *m* nu poate fi aflat din exterior servind drept etichetă *locală* pentru procesul subordonat.

Subordonarea poate fi imbricată, de exemplu

$(n{:}(m{:}P /\!/ Q) /\!/ R)$

În acest caz toate apariţiile evenimentelor implicând numele *m* sunt mascate înainte ca numele *n* să fie ataşat evenimentelor rămase, ce se află în alfabetul lui *Q* dar nu şi în al lui *P*. Nu există nici o modalitate prin care *R* să poată comunica direct cu *P* sau să ştie de existenţa lui *P* sau a numelui său *m*.

Exemple

X1 *dub*:*DUBLU*//*Q* (pentru *DUBLU* vezi 4.2 X2)

Procesul subordonat se comportă ca o simplă subrutină apelată din procesul principal *Q*. În interiorul lui *Q*, valoarea $2 \times e$ poate fi obţinută printr-o transmisie succesivă a argumentului *e* pe canalul din stânga lui *dub* şi recepţia rezultatului pe canalul din dreapta

dub.stânga!*e*→(*dub.dreapta*?*x*→...) □

X2 O subrutină poate utiliza o alta ca subordonată şi acest lucru îl face de câteva ori

4ORI=(*dub*:*DUBLU*//(μ*X.stânga*?*x*→*dub.stânga*!*x*→
dub.dreapta?*y*→*dub.stânga*!*y*→*dub.dreapta*?*z*→*dreapta*!*z*→*X*)

Această rutină însăşi este proiectată să fie utilizată ca o subrutină

4ori:*4ORI*//*Q*

Această versiune a lui *4ORI* este similară cu cea din 4.4 X1 dar nu are acelaşi efect de dublă bufferare. □

X3 O variabilă de program convenţionabilă numită *m* poate fi modelată ca un proces subordonat

m:*VAR*//*Q*

În interiorul procesului principal Q, valoarea lui m poate fi asignată, citită şi actualizată prin recepţie şi transmisie, aşa cum se descrie în 2.6.2 X2.

$m:=3;P$ este implementat prin ($m.stânga!3\rightarrow P$)
$x:=m;P$ este implementat prin ($m.dreapta?x\rightarrow P$)
$m:=m+3;P$ este implementat prin ($m.dreapta?y\rightarrow m.stânga!(y+3)\rightarrow P$)
□

Un proces subordonat poate fi folosit la implementarea unei structuri de date cu o comportare mai elaborată decât cea a unei simple variabile.

X4 (q:*BUFFER*//Q) (vezi 4.2 X9)

Procesul subordonat are rolul unei cozi nemărginite numită q. În interiorul lui Q, transmisia lui $q.stânga!v$ adaugă v la un capăt al cozii şi $q.dreapta?y$ elimină un element din celălalt capăt şi-i dă valoarea sa lui y. Dacă coada este goală, ea nu va răspunde şi sistemul se va bloca. □

X5 O stivă cu numele st este declarată

st:*STIVĂ*//Q (vezi 4.2 X10)

În interiorul procesului principal Q, $st.stânga!v$ poate fi utilizată pentru a depune valoarea v în stivă şi $st.dreapta?x$ va extrage valoarea din vârf. Pentru a putea gestiona situaţia când stiva este goală, poate fi utilizată o construcţie alternativă

($st.dreapta?x\rightarrow Q1(x)|st.gol\rightarrow Q2$)

Dacă stiva nu este goală, este selectată prima alternativă. Dacă este goală, blocajul este evitat şi este selectată a doua alternativă. □

Un proces subordonat cu mai multe canale poate fi utilizat de mai multe procese concurente în condiţiile când nu utilizează acelaşi canal.

X6 Un proces Q intenţionează să comunice un flux de valori lui R. Aceste valori trebuie bufferate de un proces buffer subordonat numit b astfel încât transmiterea de la Q să nu fie întârziată când R nu este gata pentru recepţie. Q utilizează canalul $b.stânga$ pentru transmisia sa şi R utilizează $b.dreapta$ pentru recepţia sa

(b:*BUFFER*//($Q||R$))

De remarcat că dacă R încearcă să recepționeze de la un buffer gol, sistemul nu se va bloca în mod necesar. R pur și simplu va fi întârziat până când Q va transmite următoarea valoare bufferului. (Dacă Q și R comunică cu bufferul pe *același* canal, atunci acel canal trebuie să fie în alfabetul amândorura și definiția lui || ar cere ca ele să comunice totdeauna simultan aceeași valoare – ceea ce ar fi complet greșit). □

Operatorul de subordonare poate fi folosit pentru a defini subrutine prin recursivitate. Fiecare nivel de recursivitate (cu excepția ultimului) declară o subrutină locală *nouă* care gestionează apelurile recursive în continuare.

X7 Factorial

$$FAC=\mu X.st\hat{a}nga?n\rightarrow(\textbf{if } n=0 \textbf{ then } (dreapta!1\rightarrow X)$$
$$\textbf{else } (f{:}X/\!/(f.st\hat{a}nga!(n-1)\rightarrow f.dreapta?y\rightarrow dreapta!(n\times y)\rightarrow X)))$$

Subrutina *FAC* utilizează canalele *stânga* și *dreapta* pentru a comunica parametri și rezultatele proceselor sale apelante. Ea folosește canalele *f.stânga* și *f.dreapta* pentru a comunica cu procesul subordonat numit *f*. Din această perspectivă ea este similară subrutinei *4ORI* (X2). Singura diferență este că procesul subordonat scris este izomorf lui *FAC* însuși. □

Exemplul de mai sus este un caz familiar și plictisitor de recursivitate exprimată într-un cadru notațional nefamiliar și cam complex. O idee mai puțin obișnuită este de a utiliza recursivitatea împreună cu subordonarea pentru a implementa o structură nemărginită de date. Fiecare nivel al recursivității reține o singură componentă a structurii și declară o *nouă* structură de date locală subordonată pentru a trata restul.

X8 Mulțime finită nemărginită

Un proces care implementează o mulțime recepționează membrii săi pe canalul său din stânga. După fiecare recepție, transmite un *DA* dacă a mai fost recepționată deja această valoare și *NU* altfel. Construcția este foarte similară cu mulțimea din 2.6.2 X4 cu excepția faptului că va reține mesaje de orice tip

$$MULȚIME=st\hat{a}nga?x\rightarrow dreapta!NU\rightarrow(rest{:}MULȚIME/\!/BUCLĂ(x))$$

unde $BUCLĂ(x)=\mu X.st\hat{a}nga?y\rightarrow(\textbf{if } y=x \textbf{ then } (dreapta!DA\rightarrow X)$
$$\textbf{else } (rest.st\hat{a}nga!y\rightarrow rest.dreapta?z\rightarrow dreapta!z\rightarrow X))$$

Mulțimea inițială este vidă. De aceea la recepția primului element x se va transmite *NU*. Apoi se declară un proces subordonat numit *rest* care va reține toate elementele mulțimii cu excepția lui x. *BUCLĂ* este proiectat să re-

cepţioneze elementele ulterioare ale mulţimii. Dacă noul element recepţionat este egal cu x, răspunsul *DA* este trimis înapoi imediat pe canalul din dreapta. Altfel, noul element este acceptat pentru a fi reţinut de *rest*. Se observă că oricare ar fi răspunsul (*DA* sau *NU*) trimis înapoi de *rest* se trece mai departe şi *BUCLĂ* se repetă. □

X9 Arbore binar

O reprezentare mai eficientă a unei mulţimi este aceea sub formă de arbore binar care se bazează pe o ordine totală ≤ dată între elementele sale. Fiecare nod reţine un element nou inserat şi declară *doi* arbori subordonaţi. Unul va reţine elementele mai mici decât rădăcina şi celălalt elementele mai mari. Specificarea externă a arborelui este aceeaşi ca şi în X8

$$ARBORE = stânga?x \rightarrow dreapta!NU \rightarrow (mai_mic{:}ARBORE /\!/ (mai_mare{:}ARBORE \| BUCLĂ))$$

Proiectarea lui *BUCLĂ* este lăsată ca exerciţiu. □

4.5.1 Legi

Următoarele legi evidente guvernează comunicaţiile între un proces şi subordonaţii lui. Prima lege descrie mascarea comunicaţiei în fiecare, sensul între procesul principal şi subordonat

L1A $(m{:}(c?x \rightarrow P(x))) /\!/ (m.c!v \rightarrow Q) = (m{:}P(v)) /\!/ Q$

L1B $(m{:}(d!v \rightarrow P)) /\!/ (m.d?x \rightarrow Q(x)) = (m{:}P) /\!/ Q(v)$

Dacă b este un canal neetichetat cu m, procesul principal poate comunica prin b fără afectarea subordonatului

L2 $(m{:}P /\!/ (b!e \rightarrow Q)) = (b!e \rightarrow (m{:}P /\!/ Q))$

Singurul proces capabil de a face alegerea pentru procesul subordonat este procesul principal

L3 $(m{:}(c?x \rightarrow P1(x) \mid d?y \rightarrow P2(y))) /\!/ (m.c!v \rightarrow Q) = (m{:}P1(v) /\!/ Q)$

Dacă două procese subordonate au acelaşi nume, unul dintre ele este inaccesibil

L4 $m{:}P /\!/ (m{:}Q /\!/ R) = (m{:}Q /\!/ R)$

De obicei, ordinea în care procesele subordonate sunt scrise nu contează

L5 Dacă *n* şi *m* sunt nume distincte

m:*P*//(*n*:*Q*//*R*)=*n*:*Q*//(*m*:*P*//*R*)

Utilizarea recursivităţii în definirea proceselor subordonate este suficient de surprinzătoare pentru a se ivi dubii dacă într-adevăr funţionează corect. Aceste dubii pot fi încet-încet eliminate, arătând cum progresează construcţia. Exemplul de mai jos foloseşte urma particulară a comportării unui proces şi arată cum se produce acea urmă. Mai important, se arată cum nu pot fi produse alte urme uşor diferite.

Exemplu

X1 O urmă tipică a lui *MULŢIME* este

s=<*stânga*.1,*dreapta*.*NU*,*stânga*.2,*dreapta*.*NU*>

Valoarea lui *MULŢIME*/*s* poate fi calculată într-o serie de paşi, folosind L1 şi L2

MULŢIME/<*stânga*.1>=*dreapta*!*NU*→(*rest*:*MULŢIME*//*BUCLĂ*(1))
astfel că *MULŢIME*/<*stânga*.1,*dreapta*.*NU*>=(*rest*:*MULŢIME*//*BUCLĂ*(1))
şi *MULŢIME*/<*stânga*.1,*dreapta*.*NU*,*stânga*.2>
=(*rest*:*MULŢIME*//(*rest*.*stânga*!2→*rest*.*dreapta*?*z*→*dreapta*!*z*→*BUCLĂ*(1))
=(*rest*:(*dreapta*!*NU*→(*rest*:*MULŢIME*//*BUCLĂ*(2)))
//(*rest*.*dreapta*?*z*→*dreapta*!*z*→*BUCLĂ*(1)))
=*rest*:(*rest*:*MULŢIME*//*BUCLĂ*(2))//(*dreapta*!*NU*→*BUCLĂ*(1))

De aceea *MULŢIME*/*s*=*rest*:(*rest*:*MULŢIME*//*BUCLĂ*(2))//*BUCLĂ*(1)

Este evident din cele de mai sus că

<*stânga*.1,*dreapta*.*NU*,*stânga*.2,*dreapta*.*DA*>

nu este o urmă a lui *MULŢIME*.

Cititorul poate verifica că

MULŢIME/*s*^<*stânga*.2,*dreapta*.*DA*>=*MULŢIME*/*s*
şi *MULŢIME*/*s*^<*stânga*.5,*dreapta*.*NU*>=*rest*:(*rest*:(*rest*:*MULŢIME*
//*BUCLĂ*(5))//*BUCLĂ*(2))//*BUCLĂ*(1) □

4.5.2 Diagrame de conexiune

Un proces subordonat poate fi reprezentat în interiorul unui dreptunghi reprezentând procesul care îl utilizează aşa cum se arată în 4.5 X1 fig. 4.10.

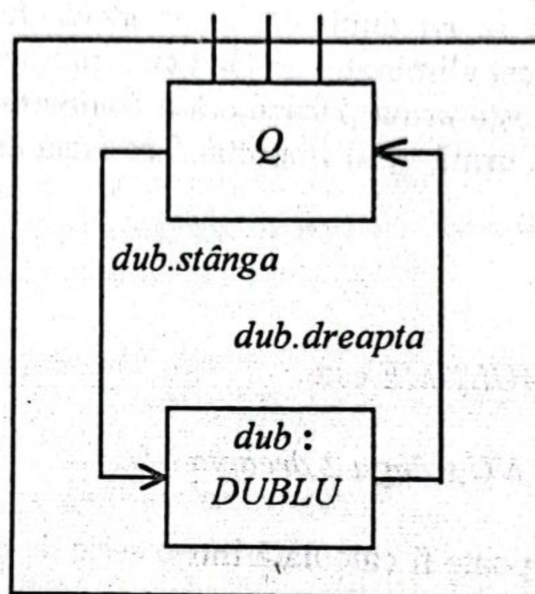


Figura 4.10

Pentru procesele subordonate imbricate, dreptunghiurile se intersectează mai mult, aşa cum decurge din 4.5 X2 şi se arată în fig. 4.11.

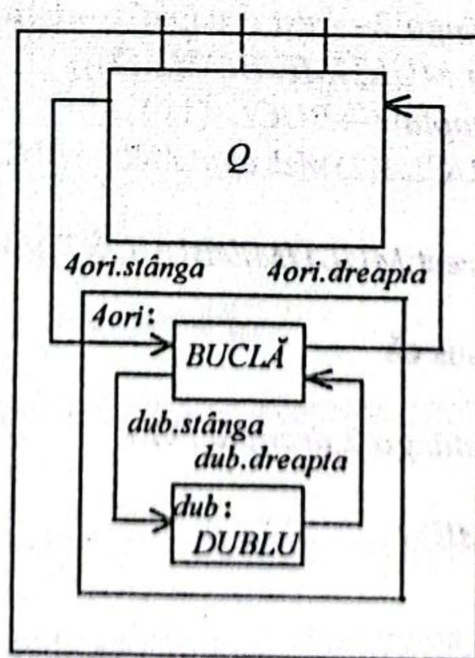
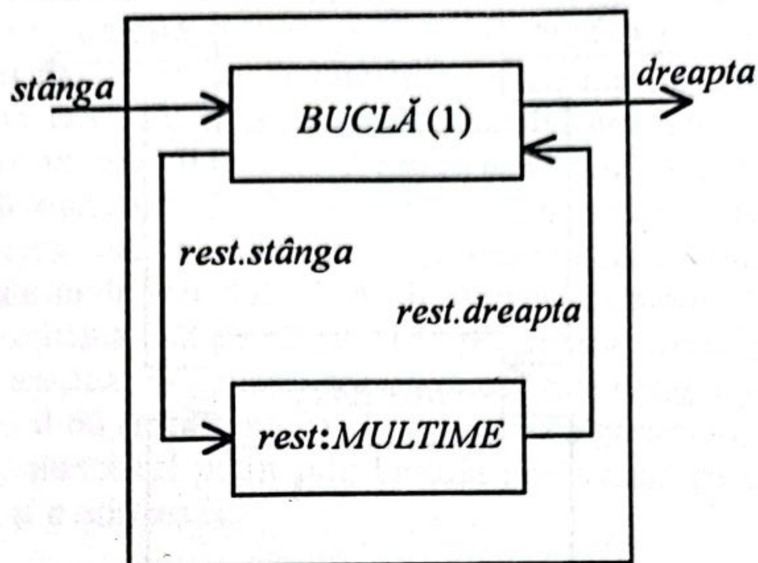


Figura 4.11

MULȚIME/⟨ *stânga*.1, *dreapta*.*NU* ⟩=



MULȚIME/ *s*=

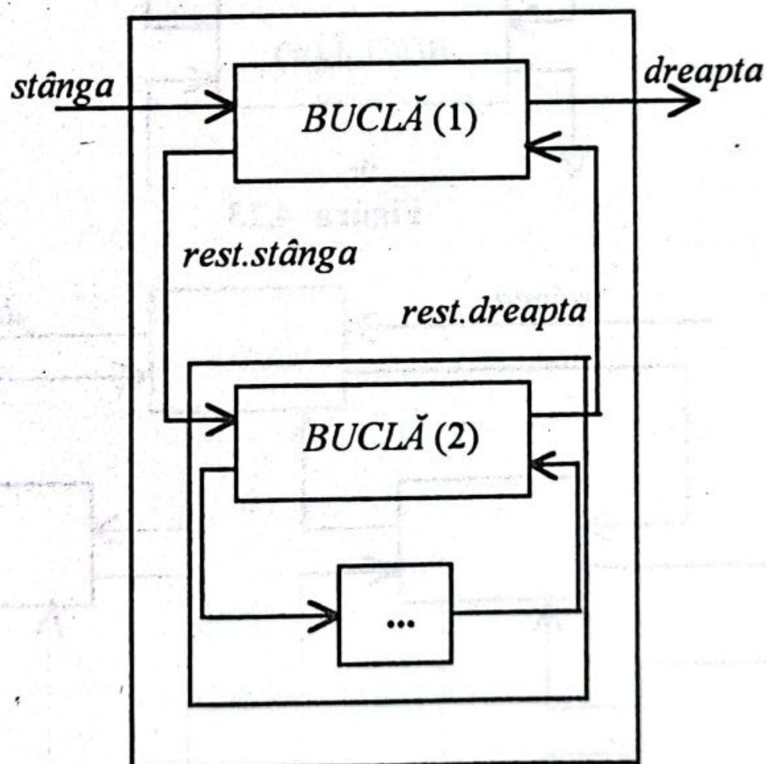


Figura 4.12

Un proces recursiv este unul imbricat cu sine însuși ca imaginea studioului unui artist, în care există pe un șevalet ultima sa pictură terminată, care arată un șevalet cu ultima sa pictură terminată..... etc. O astfel de pictură complexă în practică nu poate fi realizată. Din fericire pentru un proces, nu este necesar să fie de la început complet – creșterea se produce automat după necesitățile din timpul activității. Astfel (vezi 4.5.1 X1) putem reprezenta etapele succesive din istoria timpurie a unei mulțimi ca în fig. 4.12.

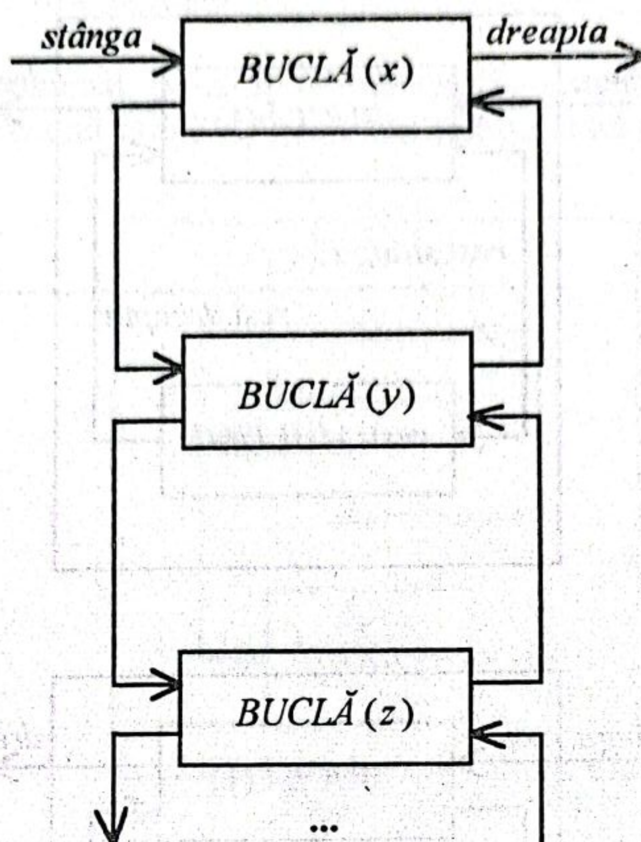


Figura 4.13

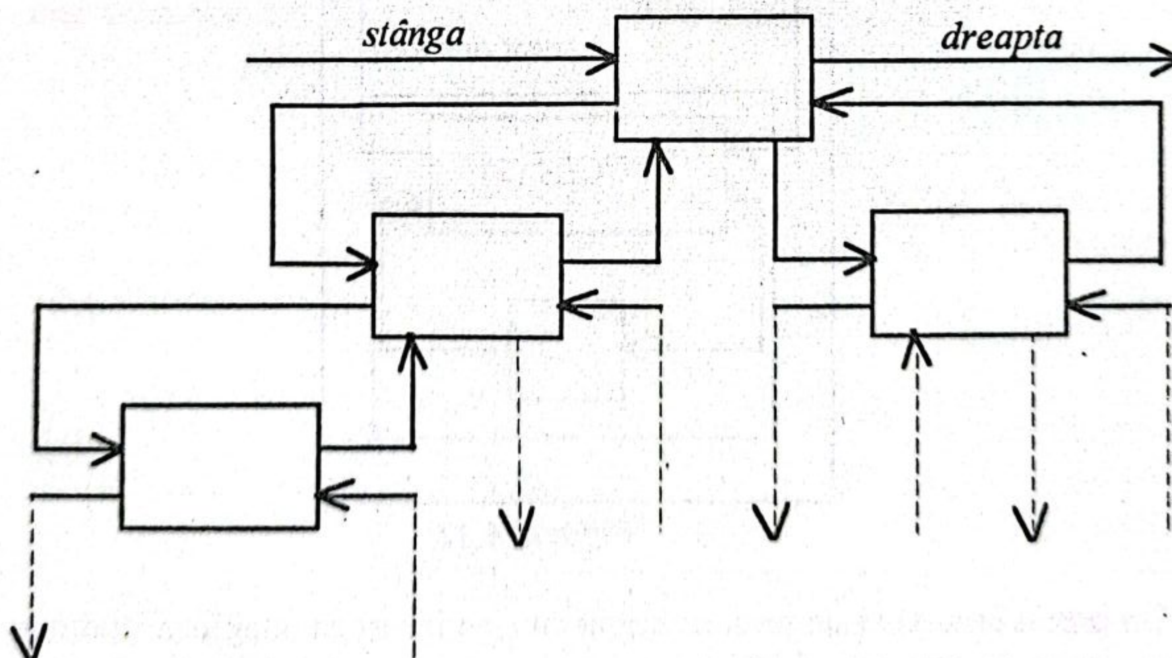


Figura 4.14

Dacă ignorăm imbricarea dreptunghiurilor, atunci putem desena o structură liniară ca în fig. 4.13. Similar, exemplul *ARBORE* (4.5 X9) ar putea fi desenat ca în fig. 4.14.

Diagramele de conexiune de mai sus sugerează cum poate fi construită o reţea de procesare corespunzătoare din componentele hardware, dreptunghiurile reprezentând circuite integrate iar arcele reprezentând legăturile fizice dintre ele. Desigur, în orice realizare practică, recursivitatea trebuie limitată (desfăşurată) până la o anumită limită finită pentru ca reţeaua fizică să opereze normal. Dacă această limită este depăşită în timpul prelucrărilor, reţeaua nu mai poate funcţiona cu succes. Realocarea şi reconfigurarea dinamică a reţelelor hardware este mult mai dificilă decât alocarea memoriei bazată pe stivă care face recursivitatea atât de eficientă în programele secvenţiale convenţionale. Cu toate acestea, în general recursivitatea este desigur justificată prin suportul pe care îl dă în inventarea şi proiectarea algoritmilor iar dacă aceasta nu este totul, atunci cel puţin prin bucuria intelectuală pe care o dă acelora care o înţeleg şi o utilizează.

5 Procese secvenţiale

5.1 Introducere

Procesul *STOP* este prin definiţie procesul care nu se angajează în nici o acţiune. Nu este un proces util şi probabil rezultă dintr-un blocaj sau altă eroare de proiectare, mai degrabă decât dintr-o alegere deliberată a proiectantului. Totuşi există un motiv ca un proces să nu se mai angajeze în nici o acţiune şi anume când a terminat de făcut ce avea de făcut. Un astfel de proces se spune că se termină cu succes. Pentru a distinge între acest proces şi *STOP* este convenabil să privim terminarea cu succes ca un eveniment special, notat cu simbolul √ (pronunţat "succes"). Un proces secvenţial este definit ca unul care are evenimentul √ în alfabetul său şi natural acesta este ultimul eveniment în care procesul se poate angaja. Impunem ca √ să nu poată fi o alternativă într-o alegere

$(x{:}B \to P(x))$ este invalidă pentru $\surd \in B$

$SKIP_A$ este definit ca un proces care nu face nimic dar se termină cu succes.

$\alpha SKIP_A = A \cup \{\surd\}$

Ca de obicei vom omite frecvent alfabetul subînţeles.

Exemple

X1 Un automat care se intenţionează să servească un client numai cu ciocolată sau bomboane şi apoi să-şi termine activitatea cu succes

$UNAV = (mon \to (choc \to SKIP \mid bonbon \to SKIP))$ □

În proiectarea unui proces care să rezolve un task complex este totdeauna util să divizăm taskul în două subtaskuri, unul din ele trebuind să se termine

cu succes înainte să înceapă un altul. Dacă P şi Q sunt procese secvenţiale cu acelaşi alfabet, compunerea lor secvenţială o notăm

$$P;Q$$

fiind un proces care întâi se comportă ca P, dar când P se termină cu succes $(P;Q)$ continuă comportându-se ca Q. Dacă P nu se termină niciodată cu succes nu o face nici $(P;Q)$.

X2 Un automat proiectat să servească doi clienţi, unul după altul

$$AVDOI=UNAV;UNAV$$ □

Un proces care repetă aceleaşi acţiuni atât cât este necesar este cunoscut ca o buclă. El poate fi definit drept un caz special de recursivitate

$$^*P=\mu X.(P;X)$$
$$=P;P;P;...$$
$$\alpha(^*P)=\alpha P-\{\surd\}$$

În mod clar, o astfel de buclă nu se va termina niciodată cu succes. De aceea este util să scoatem √ din alfabetul său.

X3 Un automat proiectat să servească orice număr de clienţi

$$AVCL={}^*UNAV$$

Acesta este identic cu *AVCL* (1.1.3 X3). □

O secvenţă de simboluri se spune că este o *propoziţie* a unui proces P dacă P se termină cu succes după angajarea în secvenţa corespunzătoare de acţiuni. Mulţimea tuturor propoziţiilor se numeşte *limbajul* acceptat de P. Astfel notaţiile introduse pentru descrierea proceselor secvenţiale pot fi folosite de asemenea pentru a defini gramatica unui limbaj simplu, astfel că ar putea deveni utile pentru comunicaţia între o fiinţă umană şi un calculator.

X4 O propoziţie din Pidgingol constă dintr-o clauză substantiv urmată de un predicat. Un predicat este un verb urmat de o clauză substantiv. Verbul este ori *muşcă* ori *zgârie*. Definiţia unei clauze substantiv este dată formal mai jos

αPIDGINGOL={un, o, pisică, câine, muşcă, zgârie}
PIDGINGOL=CLAUZĂ_SUBSTANTIV;PREDICAT
PREDICAT=VERB;CLAUZĂ_SUBSTANTIV

$VERB=(muşcă \rightarrow SKIP \mid zgârie \rightarrow SKIP)$
$CLAUZĂ_SUBSTANTIV=ARTICOL;SUBSTANTIV$
$ARTICOL=(un \rightarrow SKIP \mid o \rightarrow SKIP)$
$SUBSTANTIV=(pisică \rightarrow SKIP \mid câine \rightarrow SKIP)$

Exemple de propoziţii din Pidgingol sunt

o pisică zgârie un câine
un câine muşcă o pisică □

Pentru a descrie limbajele cu un număr infinit de propoziţii este necesar să folosim sau iteraţia sau recursivitatea.

X5 O clauză adjectiv poate conţine orice număr de adjective *mare* sau *cuminte*.

$$CLAUZĂ_SUBSTANTIV=ARTICOL;\mu X.(mare \rightarrow X \mid cuminte \rightarrow X \mid pisică \rightarrow SKIP \mid câine \rightarrow SKIP)$$

Exemple de clauze substantiv sunt

un mare mare cuminte câine
o pisică □

X6 Un proces care acceptă orice număr de *a*-uri urmate de *b*-uri şi apoi acelaşi număr de *c*-uri după care se termină cu succes

$$A^nBC^n=\mu X.(b \rightarrow SKIP \mid a \rightarrow (X;(c \rightarrow SKIP)))$$

Dacă întâi este acceptat un *b*, procesul se termină fără ca nici un *a* sau *c* să nu fie acceptat astfel că numărul lor este acelaşi. Dacă este parcursă a doua ramură, propoziţia acceptată porneşte cu *a* şi se termină cu *c* şi între aceste două evenimente este propoziţia acceptată de apelul recursiv pe procesul X. Dacă presupunem că apelul recursiv acceptă un număr egal de *a*-uri şi *c*-uri aşa va face şi apelul nerecursiv pe A^nBC^n deoarece acceptă un *a* în plus la început şi un *c* în plus la sfârşit.

Acest exemplu ne arată cum o compunere secvenţială folosită în conjuncţie cu recursivitatea poate defini o maşină cu un număr infinit de stări.

□

X7 Un proces care întâi se comportă ca A^nBC^n dar apoi acceptă un *d* urmat de acelaşi număr de *e*-uri

$$A^nBC^nDE^n=((A^nBC^n));d\rightarrow SKIP\|C^nDE^n$$

unde $C^nDE^n=F(A^nBC^n)$ cu $f(a)=c$
$f(b)=d$
$f(c)=e$

În acest exemplu, procesul din stânga lui || este responsabil pentru asigurarea unui număr egal de *a*-uri şi *c*-uri (separat de un *b*). Nu se va permite astfel un *d* până când nu a sosit acelaşi număr de *c*-uri iar *e*-urile care nu sunt în alfabetul său sunt ignorate. Procesul din dreapta lui || este responsabil pentru asigurarea unui număr egal de *e*-uri şi *c*-uri. El ignoră *a*-urile şi *b*-ul care nu sunt în alfabetul său. Perechea de procese se termină împreună când ambele şi-au completat sarcinile repartizate. □

Notaţiile pentru definirea unui limbaj cu ajutorul unui proces sunt tot atât de puternice ca şi expresiile regulate. Utilizarea recursivităţii aduce ceva din puterea gramaticilor libere de context dar nu numai. Un proces poate defini numai acele limbaje care pot fi analizate gramatical de la stânga la dreapta fără *backtracking* sau *look-ahead*. Aceasta din cauză că utilizarea operatorului alegere impune ca primul eveniment al unei alternative să fie diferit de primul eveniment al oricărei alteia. Mai mult, nu este posibil să utilizăm construcţia din X5 pentru a defini o clauză substantiv în care cuvântul *cuminte* poate fi atât un substantiv cât şi un adjectiv sau amândouă, de exemplu *un cuminte câine*, *un mare cuminte*. Utilizarea lui □ (paragraful 3.3) nu ne-ar ajuta din cauză că introduce nedeterminismul şi permite alegerea arbitrară a clauzei care va analiza restul intrării. Dacă alegerea este greşită, procesul se va bloca înaintea atingerii sfârşitului textului de intrare. Ceea ce este deci necesar pentru a rezolva această problemă este un nou tip de operator alternativă care să asigure nedeterminismul angelic ca de exemplu *sau*3 (paragraful 3.2.2). Acest nou operator impune ca două alternative să ruleze concurent până când mediul va face alegerea. Definiţia sa este lăsată ca exerciţiu.

Fără nedeterminismul angelic metoda definirii limbajului descrisă mai sus nu este tot atât de puternică ca şi gramaticile libere de context din cauză că ea necesită analiză gramaticală stânga-dreapta fără backtracking. Totuşi, introducerea lui || permite definirea limbajelor care nu sunt libere de context pentru exemplul X7.

X8 Un proces care acceptă orice întreţesere de *jos* şi *sus* cu excepţia terminării cu succes cu prima ocazie când numărul de evenimente *jos* depăşeşte numărul de evenimente *sus*

$$POZ=(jos\rightarrow SKIP\mid sus\rightarrow(POZ;POZ))$$

Dacă primul simbol este *jos* procesul *POZ* este imediat terminat. Dar dacă primul simbol este *sus*, este necesar să se accepte încă două evenimente *jos* faţă de *sus*. Singura cale de a realiza aceasta este acceptarea mai întâi a unui *jos* în plus faţă de evenimente *sus*, ulterior acceptându-se din nou un *jos* în plus faţă de un *sus*. De aceea sunt necesare două apeluri succesive recursive ale lui *POZ*, unul după altul. □

X9 Procesul M_0 se comportă ca MM_0 (1.1.4 X2)

$$M_0=(pe_loc \rightarrow M_0 \mid sus \rightarrow M_1)$$
$$M_{n+1}=POZ;M_n = \underbrace{POZ;...POZ;}_{n \text{ ori}}POZ;M_0 \qquad \text{pentru toţi } n \geq 0$$
□

Acum putem rezolva problema menţionată în 2.6.2 X3 şi întâlnită din nou în 4.5 X3, astfel că fiecare operaţie cu un proces subordonat să menţioneze explicit restul procesului utilizator care urmează după el. Efectul scontat poate fi acum mult mai convenabil realizat cu ajutorul lui *SKIP* şi a compunerii secvenţiale.

X10 Un proces *UTIL* manipulează două variabile contor numite *l* şi *m* (vezi 2.6.2 X3)

$$l{:}MM_0 \| m{:}MM_3 \| UTIL$$

Următorul proces (în interiorul lui *UTIL*) adună valoarea curentă a lui *l* la *m*

$$ADUN=(l.pe_loc \rightarrow SKIP \mid l.jos \rightarrow (ADUN;(m.sus \rightarrow l.sus \rightarrow SKIP)))$$

Dacă valoarea lui *l* este iniţial 0, nu mai trebuie făcut nimic deoarece procesul se termină. Altfel, *l* este decrementat şi valoarea actualizată (redusă) este adunată la *m* (printr-un apel recursiv la *ADUN*). Apoi *m* este este incrementat încă o dată şi *l* este de asemenea incrementat pentru a compensa decrementarea sa iniţială fiind astfel adus la valoarea iniţială. □

5.2 Legi

Legile pentru compunerea secvenţială sunt similare acelora pentru concatenare (paragraful 1.6.1), *SKIP* jucând rolul de unitate

L1 $SKIP;P=P;SKIP=P$

L2 $(P;Q);R=P;(Q;R)$
L3 $(x:B\rightarrow P(x));Q=(x:B\rightarrow(P(x);Q))$

Legea pentru operatorul alegere are corolarele

L4 $(a\rightarrow P);Q=a\rightarrow(P;Q)$
L5 $STOP;Q=STOP$

Când procesele secvenţiale se compun în paralel, combinaţia se termină cu succes numai când ambele componente sunt în această situaţie

L6 $SKIP_A||SKIP_B=SKIP_{A\cup B}$

Un proces care se termină cu succes nu mai participă în nici un eveniment oferit de un partener concurent

L7 $((x:B\rightarrow P(x))||SKIP_A)=(x:(B-A)\rightarrow(P(x)||SKIP_A))$

Ne punem întrebarea dacă o combinaţie concurentă de proces secvenţial cu proces nesecvenţial se termină cu succes. Dacă alfabetul procesului secvenţial conţine în întregime pe cel al partenerului, terminarea colaborării este determinată de procesul secvenţial, deoarece celălalt proces nu mai poate face nimic când partenerul său şi-a terminat activitatea

L8 $STOP_A||SKIP_B=SKIP_B$ dacă $\surd\notin A\wedge A\subseteq B$

Condiţia pentru validitatea acestei legi este una foarte rezonabilă care ar putea fi totdeauna observată deoarece √ este numai în alfabetul unuia din cele două procese evoluând concurent. Astfel evităm situaţia ca un proces să continue după ce s-a angajat în √.

Legile L1-L3 pot fi folosite pentru a demonstra afirmaţia din 5.1 X9 că M_0 se comportă ca MM_0 (1.1.4 X2). Aceasta se face arătând că M_n satisface mulţimea de ecuaţii recursive cu gardă folosite pentru a defini MM_n. Ecuaţia pentru M_0 este aceeaşi ca pentru MM_0

$M_0=(pe_loc\rightarrow M_0\mid sus\rightarrow M_1)$ definiţie M_0

Pentru $n>0$, trebuie să dovedim că

$M_n=(sus\rightarrow M_{n+1}\mid jos\rightarrow M_{n-1})$

Demonstraţie

$MS=POZ;M_{n-1}$	definiţie M_n
$=(jos\rightarrow SKIP \mid sus\rightarrow POZ;POZ);M_{n-1}$	definiţie POZ
$=(jos\rightarrow(SKIP;M_{n-1}) \mid sus\rightarrow(POZ;POZ);M_{n-1})$	L3
$=(jos\rightarrow M_{n-1} \mid sus\rightarrow POZ;(POZ;M_{n-1}))$	L1,L2
$=(jos\rightarrow M_{n-1} \mid sus\rightarrow POZ;M_n)$	definiţie M_n
$=MD$	definiţie M_n

Deoarece M_n se supune la aceeaşi mulţime de ecuaţii recursive cu gardă ca şi MM_n procesele sunt aceleaşi.

Această demonstraţie a fost prezentată complet pentru a ilustra utilizarea legilor şi a elimina suspiciunile provocate de transformările recurente. Ceea ce pare cel mai curios este că demonstraţia nu foloseşte inducţia după *n*. De fapt, orice încercare de a folosi inducţia după *n* va eşua din cauză că definiţia lui MM_n conţine procesul MM_{n+1}. Din fericire, aducerea aminte a legii soluţiei unice ne scoate evident din impas.

5.3 Aspecte matematice

Definiţia matematică a compunerii secvenţiale trebuie formulată astfel încât să fim siguri de adevărul legilor menţionate în paragraful anterior. O atenţie specială trebuie exercitată asupra

$$P;SKIP=P$$

Ca de obicei, tratarea proceselor deterministe este mult mai simplă şi se va face mai întâi.

5.3.1 Procese deterministe

Operaţiile cu procese deterministe sunt definite cu ajutorul urmelor rezultatelor lor. Prima şi singura acţiune a procesului *SKIP* este terminarea cu succes astfel încât el are numai două urme

L0 $urme(SKIP)=\{\langle\rangle,\langle\surd\rangle\}$

Pentru a defini compunerea secvenţială a proceselor este convenabil să definim mai întâi compunerea secvenţială a urmelor lor individuale. Dacă *s* şi *t* sunt urme şi *s* nu conţine √

$$\langle s;t\rangle=s$$
$$(s\frown\langle\surd\rangle;t)=s\frown t$$

(vezi paragraful 1.9.7. pentru detalii complete). O urmă a lui $(P;Q)$ constă dintr-o urmă a lui P şi dacă aceasta se termină cu √, evenimentul este înlocuit şi se permite generarea urmei lui Q

L1 $urme(P;Q)=\{s;t \mid s\in urme(P)\wedge t\in urme(Q)\}$

O definiţie echivalentă este

L1A $urme(P;Q)=\{s \mid s\in urme(P)\wedge\neg\langle\surd\rangle \textbf{ in } s\}$
$\cup\{s\hat{}t \mid s\hat{}\langle\surd\rangle\in urme(P)\wedge t\in urme)Q)\}$

Această definiţie este mai simplu de înţeles decât de folosit.

Scopul simbolului √ este acela de a permite terminarea procesului care se angajează în el. De aceea avem nevoie de legea

L2 $P/s=SKIP$ dacă $s\hat{}\langle\surd\rangle\in urme(P)$

Această lege este esenţială în demonstrarea lui

$P;SKIP=P$

Din nefericire, ea nu este în general adevărată. De exemplu, dacă

$P=(SKIP_{\{\}}\|c\rightarrow STOP_{\{c\}})$

atunci $urme(P)=\{\langle\rangle,\langle\surd\rangle,\langle c\rangle,\langle c,\surd\rangle,\langle\surd,c\rangle\}$ şi $P/\langle\rangle\neq SKIP$ cu toate că $\langle\surd\rangle\in urme(P)$. De aceea avem nevoie să impunem nişte constrângeri de alfabet pentru compunerea paralelă. Construcţia $(P\|Q)$ trebuie privită ca invalidă cu excepţia situaţiei

$\alpha P\subseteq\alpha Q\vee\alpha Q\subseteq\alpha P\vee\surd\in(\alpha P\cap\alpha Q\cup\overline{\alpha P}\cap\overline{\alpha Q})$

Pentru motive similare schimbarea de alfabet trebuie garantată că păstrează √ nealterat, astfel că $f(P)$ este invalidă fără

$f(\surd)=\surd$

Mai departe, dacă m este un nume de proces trebuie să adoptăm convenţia ca

$m.\surd=\surd$

În fine, nu trebuie să folosim alegerea în situaţia

$(\surd \rightarrow P \mid c \rightarrow Q)$

Această restricție se aplică și lui RUN_A când $\surd \in A$.

5.3.2 Procese nedeterministe

Compunerea secvențială a proceselor nedeterministe prezintă un număr de probleme. Prima este că un proces nedeterminist ca

$SKIP \sqcap (c \rightarrow SKIP)$

nu satisface legea L2 din paragraful anterior. O soluție ar fi să slăbim legea 5.3.1. L2 la

L2A $s\frown<\surd> \in urme(P) \Rightarrow (P/s) \sqsubseteq SKIP$

Aceasta înseamnă că ori de câte ori P se termină, o poate face fără a oferi vreun eveniment ca alternativă mediului. Pentru a menține adevărul din L2A trebuie observate toate restricțiile din paragraful anterior astfel că

SKIP nu trebuie să apară niciodată fără gardă ca operand în □
√ . nu trebuie să apară în alfabetul vreunui operand din ||

(Este posibil ca o ușoară modificare în definiția lui □ sau || să permită relaxarea acestor restricții).

În plus față de legile date mai înainte în acest capitol, compunerea secvențială a proceselor nedeterministe va satisface următoarele legi. Mai întâi, un proces divergent rămâne divergent fără a ține seama de ce este specificat să se întâmple după terminarea lui cu succes.

L1 $CHAOS;P=CHAOS$

Compunerea secvențială se distribuie cu alternativa nedeterministă

L2A $(P \sqcap Q);R=(P;R) \sqcap (Q;R)$
L2B $R;(P \sqcap Q)=(R;P) \sqcap (R;Q)$

Pentru a defini $(P;Q)$ în modelul matematic al proceselor nedeterministe (paragraful 3.9), este necesară tratarea eșecurilor și divergențelor. Dar mai întâi să descriem refuzurile (paragraful 3.4). Dacă P poate refuza X și nu se poate termina cu succes, urmează că $X \cup \{\surd\}$ este de asemenea un refuz al lui P (3.4.1 L11). În acest caz, X este un refuz al lui $(P;Q)$. Dar dacă P oferă opțiunea terminării cu succes atunci în $(P;Q)$ această acțiune poate apare indepen-

dent. Producerea ei este mascată şi orice refuz al lui Q este de asemenea un refuz al lui $(P;Q)$. Cazul când terminarea cu succes al lui P este nedeterministă este de asemenea inclus în această definiţie

D1 $$refuzuri(P;Q)=\{X \mid (X\cup\{\surd\})\in refuzuri(P)\}$$
$$\cup\{X \mid \langle\surd\rangle\in urme(P)\wedge X\in refuzuri(Q)\}$$

Urmele lui $(P;Q)$ sunt definite în aceeaşi manieră ca pentru procesele deterministe. Divergenţele lui $(P;Q)$ sunt definite cu observaţia că ansamblul diverge ori de câte ori P diverge sau când P s-a terminat cu succes şi apoi diverge Q.

D2 $$divergen\c{t}e(P;Q)=\{s \mid s\in divergen\c{t}e(P)\wedge\neg\langle\surd\rangle \textbf{ in } s\}$$
$$\cup\{s\hat{\ }t \mid s\hat{\ }\langle\surd\rangle\in urme(P)\wedge\neg\langle\surd\rangle \textbf{ in } s\wedge t\in divergen\c{t}e(Q)\}$$

Orice eşec al lui $(P;Q)$ este ori un eşec al lui P înainte ca P să se poată termina, ori un eşec al lui Q după ce P s-a terminat cu succes

D3 $$e\c{s}ecuri(P;Q)=\{(s,X) \mid (s,X\cup\{\surd\})\in e\c{s}ecuri(P)\}$$
$$\cup\{(s\hat{\ }t,X) \mid s\hat{\ }\langle\surd\rangle\in urme(P)\wedge(t,X)\in e\c{s}ecuri(Q)\}\cup\{(s,X) \mid s\in divergen\c{t}e(P;Q)\}$$

5.3.3 Implementare

SKIP este implementat ca un proces care acceptă numai simbolul "*SUCCES*. Nu contează ce face după aceea.

$$SKIP=\lambda x.\textbf{if } x=\text{"}SUCCESS \textbf{ then } STOP \textbf{ else } \text{"}BLIP$$

O compunere secvenţială se comportă ca al doilea operand dacă primul operand se termină, altfel primul operand participă în primul eveniment şi restul din el participă la compunerea cu al doilea operand.

$$\begin{aligned} secven\c{t}\breve{a}(P,Q)=\textbf{if } & P(\text{"}SUCCESS)\neq\text{"}BLIP \textbf{ then } Q \\ & \textbf{else } \lambda x.\textbf{if } P(x)=\text{"}BLIP \textbf{ then } \text{"}BLIP \\ & \qquad \textbf{else } secven\c{t}\breve{a}(P(x),Q) \end{aligned}$$

5.4 Întreruperi

În acest paragraf definim o formă de compunere secvenţială $(P\hat{\ }Q)$ care nu depinde de terminarea cu succes a lui P. Mai mult, progresul lui P este întrerupt de apariţia primului eveniment din Q, iar P nu mai este niciodată reluat.

Urmează că o urmă a lui $(P\hat{}Q)$ este chiar o urmă a lui P până la un punct, când apare întreruperea, urmată de orice urmă a lui Q

$$\alpha(P\hat{}Q)=\alpha P\cup\alpha Q$$
$$urme(P\hat{}Q)=\{s\hat{}t \mid s\in urme(P)\wedge t\in urme(Q)\}$$

Pentru a evita problemele precizăm că $\surd$ nu trebuie să fie în αP.

Următoarea lege afirmă că mediul este cel care determină când va începe Q prin selecţia unui eveniment care este oferit iniţial de Q dar nu de P

L1 $(x{:}B\rightarrow P(x))\hat{}Q=Q\square(x{:}B\rightarrow(P(x)\hat{}Q))$

Dacă $(P\hat{}Q)$ poate fi întrerupt de R aceasta este totuna cu P să poată fi întrerupt de $(Q\hat{}R)$

L2 $(P\hat{}Q)\hat{}R=P\hat{}(Q\hat{}R)$

Deoarece *STOP* nu se poate angaja în nici un eveniment nu poate fi declanşat de mediu. Similar, dacă *STOP* este interuptibil, de fapt numai întreruperea poate să apară. Astfel, *STOP* este unitatea lui ^

L3 $P\hat{}STOP=P=STOP\hat{}P$

Operatorul întrerupere execută amândoi operanzii săi cel mult o dată, astfel că se distribuie cu alternativa nedeterministă.

L4A $P\hat{}(Q\sqcap R)=(P\hat{}Q)\sqcap(P\hat{}R)$

L4B $(Q\sqcap R)\hat{}P=(Q\hat{}P)\sqcap(R\hat{}P)$

În fine, nimeni nu poate schimba natura unui proces divergent prin întrerupere, nici nu este prea sigură specificarea unui proces divergent după întrerupere

L5 $CHAOS\hat{}P=P\hat{}CHAOS=CHAOS$

În restul acestui paragraf vom insista pe faptul că evenimentele iniţiale posibile ale proceselor care forţează întreruperea nu există în alfabetul procesului întrerupt. Deoarece apariţia întreruperii este vizibilă şi controlabilă de mediu, această restricţie păstrează determinismul şi efectele asupra operatorilor sunt simplificate. Pentru a evidenţia conservarea determinismului extindem definiţia operatorului alegere. În condiţiile când $c\notin B$

$(x{:}B \rightarrow P(x) \mid c \rightarrow Q)$ este prescurtarea pentru

$(x{:}(B \cup \{c\}) \rightarrow (\textbf{if } x{=}c \textbf{ then } Q \textbf{ else } P(x)))$

şi similar pentru mai mulţi operanzi.

5.4.1 Catastrofa

Fie $\lightning$ simbolul pentru un eveniment de întrerupere catastrofic, pe care este rezonabil să-l presupunem că n-ar fi cauzat de P. Mai formal

$$\lightning \notin \alpha P$$

Atunci un proces care se comportă ca P până la catastrofă şi apoi ca Q este definit

$$P \lightning Q = P \,\hat{}\, (\lightning \rightarrow Q)$$

Aici Q este un proces cu posibilitatea de a se reface după catastrofă. De notat că operatorul $\hat{\lightning}$ este deosebit de evenimentul $\lightning$. Prima lege este chiar o formalizare evidentă a descrierii operatorului

L1 $(P \hat{\lightning} Q)/(s \,\hat{}\, \langle \lightning \rangle) = Q$ pentru $s \in urme(P)$

În modelul determinist, această singură lege defineşte unic înţelesul operatorului. Într-un univers nedeterminist, unicitatea ar necesita legi adiţionale afirmând stricteţea şi distributivitatea în ambele argumente.

A doua lege dă o descriere mai explicită a primului şi următorilor paşi ai procesului, arătându-ne cum se distribuie $\hat{\lightning}$ înapoi cu $\rightarrow$

L2 $(x{:}B \rightarrow P(x)) \hat{\lightning} Q = (x{:}B \rightarrow (P(x) \hat{\lightning} Q \mid \lightning \rightarrow Q)$

Această lege defineşte în mod unic operatorul în procesele deterministe.

5.4.2 Restart

Un posibil răspuns la catastrofă este reluarea procesului original. Fie P un proces astfel ca $\lightning \notin \alpha P$. Specificăm $\hat{P}$ ca un proces care se comportă ca P până ce apare $\lightning$ şi după fiecare eveniment $\lightning$ se comportă ca P de la început Un astfel de proces este numit *restartabil* şi este definit de recursivitatea simplă

$$\alpha \hat{P} = \alpha P \cup \{↯\}$$
$$\hat{P} = \mu X.(P ↯ X)$$
$$= P ↯ (P ↯ (P ↯ \ldots))$$

Aceasta este o recursivitate cu gardă deoarece apariţia lui X este cu gardă datorită lui ↯. Procesul $\hat{P}$ este desigur un proces ciclic (paragraful 1.8.3) chiar dacă P nu este. Catastrofa nu este singurul motiv pentru restart. Considerăm un proces proiectat să prezinte un joc interacţionând cu utilizatorul uman prin intermediul selecţiei tastelor unei tastaturi (vezi descrierea funcţiei *interact* din paragraful 1.4). Oamenii se plictisesc uneori uşor de un joc şi vor să înceapă un altul. Pentru acest motiv este prevăzută o tastă nouă, specială, pe tastatură. Apăsarea ei în orice moment al jocului va restarta jocul. Este convenabil să definim un joc P independent de facilitatea de restartare şi apoi să-l transformăm într-un joc $\hat{P}$ utilizând operatorul de mai sus. Această idee se datoreşte lui Alex Teruel.

Definiţia formală a lui $\hat{P}$ este exprimată de legea

L1 $\hat{P} / s^\frown\langle ↯ \rangle = \hat{P}$ pentru $s \in urme(P)$

Dar această lege nu defineşte unic $\hat{P}$ deoarece ea este satisfăcută şi de RUN. Totuşi, P este cel mai mic proces determinist care satisface L1.

5.4.3 Alternarea

Presupunem că P şi Q sunt procese care joacă jocuri în maniera descrisă în paragraful 5.4.2. Un om doreşte să joace ambele jocuri simultan, alternând între ele la fel cum un maestru de şah joacă un simultan ciclând succesiv pe la partenerii mai slabi. De aceea prevedem o nouă tastă ⊗ care determină alternarea între cele două jocuri P şi Q. Aceasta este similar unei întreruperi prin aceea că jocul curent este întrerupt într-un punct arbitrar, dar diferă de întrerupere prin faptul că starea curentă a jocului curent este salvată astfel că poate fi reluat când celălalt joc este întrerupt la un moment dat. Procesul care descrie jocul P şi Q simultan în acest mod este notat $(P \otimes Q)$ şi este cel mai clar specificat de legile

L1 $\otimes \in (\alpha(P \otimes Q) - \alpha P - \alpha Q)$

L2 $(P \otimes Q) / s = (P / s) \otimes Q$ dacă $s \in urme(P)$

L3 $(P \otimes Q) / \langle \otimes \rangle = (Q \otimes P)$

Dorim să aflăm cel mai mic proces care satisface L2 şi L3. O descriere mai constructivă a operatorului poate fi derivată din aceste legi, arătându-ne cum ⊗ se distribuie înapoi cu →

L4 $(x{:}B \to P(x)) \otimes Q = (x{:}B \to (P(x) \otimes Q) \mid \otimes \to (Q \otimes P))$

Operatorul alternare este util nu numai pentru jocuri. O facilitate asemănătoare ar putea fi asigurată într-un sistem de operare "prietenos" cu facilităţi de alternare a utilităţilor. De exemplu nu dorim să pierdem poziţia în editor când apelăm un program "help" şi nici viceversa.

5.4.4 Puncte de control

Fie P un proces care descrie comportarea unui sistem de baze de date pe o perioadă lungă de timp. Când apare (↯) una din cele mai dificile situaţii ar fi ca P să repornească din starea iniţială, pierzând toate datele multiple acumulate de sistem. Ar fi mult mai convenabil să ne reîntoarcem la o stare mai recentă a sistemului care se ştie că este satisfăcătoare. O astfel de stare este cunoscută ca punct de control (checkpoint). De aceea afectăm o nouă tastă © care ar putea fi apăsată numai când starea curentă a sistemului este cunoscută ca fiind satisfăcătoare. Când apare ↯, este restaurat cel mai recent punct de control. Dacă nu există nici unul, se restaurează starea iniţială. Presupunem că © şi ↯ nu sunt în alfabetul lui P şi definim $Pc(P)$ un proces care se comportă ca P dar răspunde într-o manieră adecvată acestor două evenimente.

Definiţia formală a lui $Pc(P)$ este cel mai pe scurt formalizată de legile

L1 $Pc(P)/(s^\frown\langle\text{↯}\rangle) = Pc(P)$ pentru $s \in urme(P)$

L2 $Pc(P)/s^\frown\langle\text{©}\rangle = Pc(P/s)$ pentru $s \in urme(P)$

$Pc(P)$ poate fi definit mai explicit în termenii unui operator binar $Pc2(P,Q)$, unde P este procesul curent şi Q este cel mai recent punct de control aşteptând să fie reluat. Dacă catastrofa apare înainte de primul punct de control, sistemul reporneşte aşa cum se descrie în legile

L3 $Pc(P) = Pc2(P,P)$

L4 Dacă $P = (x{:}B \to P(x))$

atunci $Pc2(P,Q) = (x{:}B \to Pc2(P(x),Q) \mid \text{↯} \to Pc2(Q,Q) \mid \text{©} \to Pc2(P,P))$

Legea L4 este sugestivă pentru o metodă de implementare practică în care starea din punctul de control este reţinută pe un suport ieftin dar durabil ca de exemplu disc magnetic sau bandă. Când apare evenimentul ©, starea curentă este copiată ca noul punct de control. Când apare ↯, punctul de control este copiat ca noua stare curentă. Pentru motive de economie, un implementa-

tor de sistem se asigură de faptul că datele sunt cât mai mult partajate între starea curentă şi stările din punctele de control. O astfel de optimizare este foarte dependentă de maşină şi aplicaţie. Iată pentru ce este aşa de plăcută şi simplă matematica.

Operatorul de punct de control este util nu numai pentru sisteme baze de date pe scară mare. Când jucăm un joc dificil putem dori să explorăm o anumită direcţie de joc fără a intra în ea. Astfel se apasă © pentru a reţine poziţia curentă şi dacă explorările sunt fără succes, folosirea tastei va restabili starea din punctul de control.

Aceste idei despre puncte de control au fost explorate de Ian Hayes.

5.4.5 Puncte de control multiple

Când utilizăm un sistem cu puncte de control $Pc(P)$ se poate întâmpla ca un punct de control să genereze eroare. În astfel de cazuri este de dorit să anulăm cel mai recent punct de control şi să mergem la anteriorul. Pentru aceasta avem nevoie de un sistem care să reţină două sau mai multe din cele mai recente stări din puncte de control corespunzătoare. În principiu, nu există nici un motiv pentru care n-am defini un sistem $Pcm(P)$, care reţine în timp toate punctele de control de la început. Fiecare apariţie a lui ↯ returnează starea ce era chiar *înaintea* celui mai recent © şi nu starea de după el. Ca întotdeauna insistăm

$$\alpha Pcm(P)-\alpha P=\{©,↯\}$$

Un ↯ înaintea unui © ne întoarce la început

L1 $Pcm(P)/s\hat{}\langle ↯\rangle=Pcm(P)$ pentru $s\in urme(P)$

Un ↯ după un © anulează efectul a tot ce s-a întâmplat până atunci *incluzând* cel mai recent ©.

L2 $Pcm(P)/s\hat{}\langle ©\rangle\hat{}t\hat{}\langle ↯\rangle=Pcm(P)/s$ pentru $(s\upharpoonright\alpha P)\hat{}t\in urme(P)$

O descriere mai explicită a lui $Pcm(P)$ poate fi dată în termenii unui operator binar $Pcm2(P,Q)$, unde P este procesul curent şi Q este format dintr-o stivă de puncte de control aşteptând să fie reluate dacă e necesar. Conţinutul iniţial al stivei este o secvenţă infinită de copii ale lui P

L3 $Pcm(P)=\mu X.Pcm2(P,X)$
$=Pcm2(P,Pcm(P))$
$=Pcm2(P,Pcm2(P,Pcm2(P,\ldots)))$

La apariția lui ©, starea curentă este salvată iar la apariția lui ↯ întreaga stivă este actualizată

L4 Dacă $P=(x{:}B\rightarrow P(x))$
atunci $Pcm2(P,Q)=(x{:}B\rightarrow Pcm2(P(x),Q)$
$\quad | \text{©}\rightarrow Pcm2(P,Pcm2(P,Q))$
$\quad | ↯\rightarrow Q)$

Modelul de recursivitate care apare în L4 este chiar ingenios dar facilitatea de puncte de control multiple ar putea fi foarte scumpă de implementat în practică când numărul lor crește mult.

5.4.6 Implementare

Implementarea diferitelor versiuni de întrerupere se bazează pe legile care arată cum se distribuie operatorii cu →. Să considerăm de exemplu operatorul alternanță (5.4.3 L4)

alternanță(P,Q)=
  λ*x*.**if** *x*=⊗ **then** *alternanță(Q,P)*
    else if *P*(*x*)="*BLIP* **then** "*BLIP*
      eise *alternanță*(*P*(*x*),*Q*)

O implementare mai surprinzătoare este aceea a lui *Pcm* (5.4.5 L3, L4)

Pcm(*P*)=*Pcm*2(*P*,*Pcm*(*P*))

unde *Pcm*2(*P*,*Q*)=
  λ*x*.**if** *x*=↯ **then** *Q*
    else if *x*=© **then** *Pcm*2(*P*,*Pcm*2(*P*,*Q*))
      else if *P*(*x*)="*BLIP* **then** "*BLIP*
        else *Pcm*2(*P*(*x*),*Q*)

Când această funcție este executată, numărul respectiv dimensiunea datelor salvate crește proporțional cu numărul de puncte de control, astfel că spațiul de depunere disponibil este foarte repede consumat. Desigur, spațiul de depunere poate fi refăcut de un program special de colectare a zonelor disponibile (garbage collector) la fiecare apariție a lui ↯ dar nici aceasta nu este de prea mare folos. Ca și în cazul altor recursivități, constrângerile implementării practice determină o margine finită în adâncime. În acest caz, proiectantul ar putea impune o limită a numărului de puncte de control reținute, renunțând la primele din ele. Dar o astfel de metodă de programare nu se poate exprima recursiv prea elegant.

5.5 Asignare

În acest paragraf vom introduce cele mai importante aspecte ale programării secvenţiale convenţionale, anume asignările, condiţiile şi buclele. Pentru a simplifica formularea legilor utile vor fi definite câteva notaţii mai speciale.

Principala caracteristică a programării convenţionale este asignarea. Dacă x este o variabilă de program, e este o expresie şi P un proces

$$(x{:=}e;\ P)$$

este un proces care se comportă ca P cu deosebirea că valoarea iniţială a lui x se defineşte a fi valoarea iniţială a expresiei e. Valorile iniţiale ale celorlalte variabile sunt neschimbate. Asignarea însăşi poate fi definită

$$(x{:=}e)=(x{:=}e;\ SKIP)$$

Asignarea simplă se generalizează uşor la asignare multiplă. Fie x o listă de variabile distincte

$$x=x_0,x_1,\ldots x_{n-1}$$

Fie e o altă listă de expresii

$$e=e_0,e_1,\ldots e_{n-1}$$

În condiţiile în care lungimile celor două liste sunt egale

$$x{:=}e$$

asignează valoarea iniţială a lui e_i la x_i pentru toţi i. De notat că *toţi* e_i sunt evaluaţi înaintea *oricărei* asignări astfel că dacă variabila y apare în expresia g

$$y{:=}f;\ z{:=}g$$

atunci ea este diferită de

$$y,z{:=}f,g$$

Fie b o expresie a cărei evaluare este logică (*true* sau *false*). Dacă P şi Q sunt procese

$P \nless b \ngtr Q$ (*P* dacă *b* altfel *Q*)

este un proces care se comportă ca P dacă valoarea iniţială a lui b este *true* sau ca Q dacă valoarea iniţială a lui b este *false*. Notaţia este imediată dar mai puţin stânjenitoare ca cea tradiţională

if b **then** P **else** Q

Pentru motive similare bucla tradiţională

while b do Q

va fi scrisă

$b*Q$

Aceasta poate fi definită prin recursivitate

D1 $b*Q=\mu X.((Q;X)\ \triangleleft b \triangleright\ SKIP)$

Exemple

X1 Un proces care se comportă ca MM_n (1.1.4 X2)

$$X1=\mu X.(pe_loc\rightarrow X \mid sus\rightarrow(n:=1;X))$$
$$\triangleleft n=0 \triangleright$$
$$(sus\rightarrow(n:=n+1;X) \mid jos\rightarrow(n:=n-1;X))$$

Valoarea curentă a contorului este înregistrată în variabila n □

X2 Un proces care se comportă ca MM_0

$n:=0; X1$

Valoarea iniţială a contorului este pusă pe zero □

X3 Un proces care se comportă ca *POZ* (5.1 X8)

$n:=1; (n>0)*(sus\rightarrow n:=n+1 \mid jos\rightarrow n:=n-1)$

Recursivitatea a fost înlocuită cu o buclă convenţională □

X4 Un proces care divide numărul natural x la numărul natural y, asignând câtul la q şi restul la r

$CÂT=(q:=x\div y;\ r:=x-q\times y)$ □

X5 Un proces cu acelaşi efect ca şi X4 care calculează câtul prin metoda lentă a scăderii repetate

$$CÂTLUNG=(q:=0;\ r:=x;\ ((r\geq y)*(q:=q+1;\ r:=r-y))) \qquad \square$$

În exemplul anterior (4.5 X3) am arătat cum comportarea unei variabile poate fi modelată printr-un proces subordonat care-şi comunică valoarea procesului care-l foloseşte. În acest capitol am lăsat special la o parte această tehnică pentru că nu are proprietăţile matematice pe care le-am dori. De exemplu, dorim

$$(m:=1;\ m:=1)=(m:=1)$$

dar din nefericire

$$(m.stânga!1\rightarrow m.stânga!1\rightarrow SKIP)\neq(m.stânga!1\rightarrow SKIP)$$

5.5.1 Legi

În legile de asignare, x şi y semnifică liste de variabile distincte. În continuare, e, $f(x)$, $f(e)$ semnifică liste de expresii ce conţin posibile apariţii ale variabilelor din listele x sau y, iar $f(e)$ conţine e_i dacă $f(x)$ conţine x_i pentru toţi indicii i. Pentru simplitate vom presupune că toate expresiile dau întotdeauna un rezultat pentru orice valoare a variabilelor pe care le conţin

L1 $(x:=x)=SKIP$
L2 $(x:=e;\ x:=f(x))=(x:=f(e))$
L3 dacă x şi y este o listă de variabile distincte $(x:=e)=(x,y:=e,y)$
L4 dacă x,y,z sunt de aceeaşi lungime ca e,f,g, respectiv
$(x,y,z:=e,f,g)=(x,z,y:=e,g,f)$

Folosind aceste legi este posibil de transformat orice secvenţă de asignări într-una singură, o listă a tuturor variabilelor implicate.

Dacă $\lhd b\rhd$ este considerat un operator binar infix el posedă câteva proprietăţi algebrice familiare

L5-6 $\lhd b\rhd$ este idempotent, asociativ şi se distribuie cu $\lhd c\rhd$
L7 $P\lhd true\rhd Q=P$
L8 $P\lhd false\rhd Q=Q$
L9 $P\lhd\neg b\rhd Q=Q\lhd b\rhd P$
L10 $P\lhd b\rhd(Q\lhd b\rhd R)=P\lhd b\rhd R$
L11 $P\lhd(a\lhd b\rhd c)\rhd Q=(P\lhd a\rhd Q)\lhd b\rhd(P\lhd c\rhd Q)$

L12 $x := e;\ (P \triangleleft b(x) \triangleright Q) = (x := e;\ P) \triangleleft b(e) \triangleright (x := e;\ Q)$
L13 $(P \triangleleft b \triangleright Q);\ R = (P;\ R) \triangleleft b \triangleright (Q;\ R)$

Pentru a testa efectiv asignarea în procese concurente este necesar să impunem restricția ca nici o variabilă asignată într-un proces concurent să nu fie folosită în altul. Pentru a întări această restricție introducem două noi categorii de simboluri în alfabetele proceselor secvențiale.

$var(P)$ mulțimea variabilelor care pot fi asignate în interiorul lui P
$acc(P)$ mulțimea variabilelor care pot fi accesate în expresii din P

Toate variabilele care pot fi schimbate pot fi de asemenea accesate

$$var(P) \subseteq acc(P) \subseteq \alpha P$$

Similar, definim $acc(e)$ mulțimea variabilelor ce apar în e. Astfel dacă P și Q sunt combinate prin || trebuie ca

$$var(P) \cap acc(Q) = acc(P) \cap var(Q) = \{\}$$

Odată această condiție îndeplinită, nu mai contează dacă o asignare are loc înainte de combinarea paralelă sau în interiorul unui proces după ce acesta începe să activeze concurent.

L14 $((x := e;\ P) \| Q) = (x := e;\ (P \| Q))$

în condițiile când $x \subseteq var(P) - acc(Q)$ și $acc(e) \cap var(Q) = \{\}$

O consecință imediată a acesteia este

$$(x := e;\ P) \| (y := f;\ Q) = (x, y := e, f;\ (P \| Q))$$

în condițiile când $x \subseteq var(P) - acc(Q) - acc(f)$
și $y \subseteq var(Q) - acc(P) - acc(e)$.

Cele de mai sus ne arată că restricțiile de alfabet protejează asignările dintr-un proces component al perechii concurente să nu poată interfera cu asignările din celălalt. Într-o implementare, secvențe de asignări pot fi executate atât împreună cât și întrețesut fără a le diferenția de acțiunile externe observabile ale procesului.
În fine, compunerea concurentă se distribuie cu operatorul condiție

L15 $P \| (Q \triangleleft b \triangleright R) = (P \| Q) \triangleleft b \triangleright (P \| R)$

când $acc(b) \cap var(P)=\{\}$.

Această lege afirmă încă o dată că nu contează dacă b este evaluat înainte sau după compunerea paralelă.

Vom trata acum problema care se naşte când expresiile sunt nedefinite pentru anumite valori ale variabilelor ce le conţin. Dacă e este o listă de expresii, definim $\mathcal{D}e$ ca o expresie booleană care este adevărată când toţi operanzii lui e sunt în domeniile operatorilor lor. De exemplu, pentru numere naturale

$$\mathcal{D}(x \div y)=(y>0)$$
$$\mathcal{D}(y+1, z+y)=true$$
$$\mathcal{D}(e+f)=\mathcal{D}e \wedge \mathcal{D}f$$
$$\mathcal{D}(r-y)=y \leq r$$

Este rezonabil de insistat ca $\mathcal{D}e$ să fie totdeauna definit

$$\mathcal{D}(\mathcal{D}e)=true$$

În mod deliberat am lăsat complet nespecificat rezultatul unei încercări de a evalua expresii nedefinite – unde se poate întâmpla de fapt orice. Aceasta este exemplificat de utilizarea lui $CHAOS$ în următoarele legi

L16 $(x:=e)=(x:=e \lhd \mathcal{D}e \rhd CHAOS)$

L17 $P \lhd b \rhd Q=((P \lhd b \rhd Q) \lhd \mathcal{D}b \rhd CHAOS)$

Mai mult, legile L2, L5 şi L12 necesită uşoare modificări.

L2' $(x:=e;\ x:=f(x))=(x:=f(e) \lhd \mathcal{D}e \rhd CHAOS)$

L5' $(P \lhd b \rhd P)=(P \lhd \mathcal{D}b \rhd CHAOS)$

5.5.2 Specificaţii

O specificare a unui proces secvenţial descrie nu numai urmele evenimentelor care au loc dar şi relaţia între aceste urme, valorile iniţiale ale variabilelor programului şi valorile lor finale. Pentru indicarea valorii iniţiale a unei variabile de program x vom utiliza chiar numele ei x. Pentru a indica valoarea finală adăugăm la x semnul $\surd$, ca de exemplu $x^{\surd}$. Valoarea lui $x^{\surd}$ nu este observabilă până ce procesul nu se termină sau ultimul eveniment al urmei este $\surd$

Acest lucru face să nu specificăm nimic despre $x^{\surd}$, ci numai $\overline{ur_0}=\surd$.

Exemple

X1 Un proces care nu face nici o acţiune, dar adună 1 valorii lui x şi se termină cu succes cu valoarea lui y neschimbată

$$ur=<>\vee(ur=<\surd>\wedge x^{\surd}=x+1\wedge y^{\surd}=y)$$ □

X2 Un proces care se angajează într-un eveniment al cărui simbol este valoarea iniţială a unei variabile x şi apoi se termină cu succes lăsând valorile finale ale lui x şi y egale cu cele iniţiale.

$$ur=<>\vee ur=<x>\vee(ur=<x,\surd>\wedge x^{\surd}=x\wedge y^{\surd}=y)$$ □

X3 Un proces care memorează identitatea primului său eveniment ca valoarea finală a lui x

$$\#ur\leq 2\wedge(\#ur=2\Rightarrow(ur=<x^{\surd},\surd>\wedge y^{\surd}=y))$$ □

Funcţionarea corectă a unui proces depinde adesea de precondiţiile $S(x)$ ale valorilor iniţiale pentru variabilele de program x. Acestea pot fi exprimate scriind $S(x)$ ca o ante–specificaţie.

X4 Un proces care divide un număr nenegativ x la unul pozitiv y şi asignează câtul lui q şi restul lui r

$$DIV=(y>0\Rightarrow ur=<>\vee(ur=<\surd>\wedge q^{\surd}=(x\div y)\wedge r^{\surd}=x-(q^{\surd}\times y)\wedge y^{\surd}=y\wedge x^{\surd}=x))$$

Fără precondiţie această specificaţie ar fi fost în general imposibilă. □

X5 Iată câteva specificaţii mai complexe care vor fi folosite mai târziu.

$$BUCLĂDIV=(ur=<>\vee(ur=<\surd>\wedge r=(q^{\surd}-q)\times y+r^{\surd}\wedge r^{\surd}<y\wedge x^{\surd}=x\wedge y^{\surd}=y))$$
$$T(n)=r<n\times y$$ □

Toate variabilele din aceste specificaţii şi următoarele se referă la numere naturale astfel că scăderea nu e definită dacă al doilea operand este mai mare ca primul.

Vom formula acum legile care pun în evidenţă că un proces îşi satisface specificaţiile. Fie $S(x,ur,x^{\surd})$ o specificaţie. Pentru a dovedi că *SKIP* satisface această specificaţie în mod clar, specificaţia trebuie să fie adevărată când urma este vidă. Mai mult, ea trebuie să fie adevărată când urma este <√> şi valorile

finale ale tuturor variabilelor $x^{\surd}$ sunt egale cu valorile lor iniţiale. Aceste două condiţii sunt de asemenea suficiente aşa cum se afirmă în următoarea lege

L1 Dacă $S(x,\langle\rangle,x^{\surd})$
şi $S(x,\langle\surd\rangle,x)$
atunci *SKIP* **sat** $S(x,ur,x^{\surd})$

X6 Cea mai puternică specificare satisfăcută de *SKIP* este

$$SKIP_A \textbf{ sat } (ur=\langle\rangle \vee (ur=\langle\surd\rangle \wedge x^{\surd}=x))$$

unde x este o listă a tuturor variabilelor din A iar $x^{\surd}$ este o listă a variabilelor cu √. X6 este o consecinţă imediată a lui L1 şi viceversa. □

X7 *SKIP* **sat** $(r<y \Rightarrow (T(n+1) \Rightarrow BUCLĂDIV))$
Demonstraţie
(1) Înlocuind *ur* cu <> în specificaţie avem

$$r<y \wedge T(n+1) \Rightarrow \langle\rangle=\langle\rangle \vee \ldots$$

care este o tautologie.
(2) Înlocuind *ur* cu <√> şi valorile finale cu cele iniţiale avem

$$r<y \wedge T(n+1) \Rightarrow (\langle\surd\rangle=\langle\rangle \vee (\langle\surd\rangle=\langle\surd\rangle \wedge x=x \wedge y=y \wedge r=((q-q)\times y+r) \wedge r<y))$$

care este o teoremă banală. Acest rezultat va fi folosit în X10. □

O precondiţie a asignării cu succes $x:=e$ este ca expresia e să fie definită. În acest caz, dacă P satisface o specificaţie $S(x)$, $(x:=e;\ P)$ satisface aceeaşi specificaţie după modificarea care reflectă că valoarea iniţială a lui x este e.

L2 Dacă P **sat** $S(x)$
atunci $(x:=e;\ P)$ **sat** $(\mathcal{D}e \Rightarrow S(e))$

Legea pentru asignarea simplă poate fi derivată din L2 înlocuind P cu *SKIP* şi folosind X6 şi 5.2 L1.

L2A $x_0:=e$ **sat** $(\mathcal{D}e \wedge ur\neq\langle\rangle \Rightarrow ur=\langle\surd\rangle \wedge x_0^{\surd}=e \wedge x_1^{\surd}=x_1 \wedge \ldots)$

O consecinţă a lui L2A este aceea că pentru orice P, cea mai tare afirmaţie care se poate demonstra despre $(x:=1/0;\ P)$ este

$(x{:}1/0;P)$ **sat** *true*

Oricare ar fi dezideratul concret la care vrem să ajungem el nu poate fi atins pornind de la o asignare ilegală.

Exemple

X8 $SKIP$ **sat** $(ur\neq\langle\rangle\Rightarrow ur=\langle\surd\rangle\wedge q^{\surd}=q\wedge r^{\surd}=r\wedge y^{\surd}=y\wedge x^{\surd}=x)$
de aceea $(r:=x-q\times y;\ SKIP)$ **sat** $(x\geq q\times y\wedge ur\neq\langle\rangle\Rightarrow$
$ur=\langle\surd\rangle\wedge q^{\surd}=q\wedge r^{\surd}=(x-q\times y)\wedge y^{\surd}=y\wedge x^{\surd}=x)$
de aceea $(q:=x\div y;\ r:=x-q\times y)$ **sat** $(y>0\wedge x\geq(x\div y)\times y\wedge ur\neq\langle\rangle\Rightarrow$
$ur=\langle\surd\rangle\wedge q^{\surd}=(x\div y)\wedge r^{\surd}=(x-(x\div y)\times y)\wedge y^{\surd}=y\wedge x^{\surd}=x)$

Specificarea din ultima linie este echivalentă lui DIV' care a fost definit în X4.
□

X9 Presupunem că X **sat** $(T(n)\Rightarrow BUCLĂDIV)$
de aceea $(r:=r-y;\ X)$ **sat** $(y\leq r\Rightarrow(r-y<n\times y\Rightarrow(ur=\langle\rangle\vee ur=\langle\surd\rangle\wedge(r-y)=\ \ldots\ )))$
deci $(q:=q+1;\ r:=r-y;\ X)$ **sat** $(y\leq r\Rightarrow(r<(n+1)\times y\Rightarrow BUCLĂDIV'))$
unde $BUCLĂDIV'=(ur=\langle\rangle\vee ur=\langle\surd\rangle\wedge(r-y)=(q^{\surd}-(q+1))\times y+r^{\surd}$
$\wedge r^{\surd}<y\wedge x^{\surd}=x\wedge y^{\surd}=y))$

Din algebra elementară a numerelor naturale

$y\leq r\Rightarrow(BUCLĂDIV'\equiv BUCLĂDIV)$

de aceea $(q:=q+1;\ r:=r-y;\ X)$ **sat** $(y\leq r\Rightarrow(T(n+1)\Rightarrow BUCLĂDIV))$
Acest rezultat va fi folosit în X10. □

Pentru compunerea secvenţială generală este necesară o lege mult mai complicată în care urmele componentelor sunt compuse secvenţial şi starea iniţială a celei de-a doua componente este identică cu starea finală a primeia. Totuşi, valorile variabilelor în această stare intermediară nu sunt observabile, fiind asigurată numai existenţa unor astfel de valori

L3 Dacă P **sat** $S(x,ur,x^{\surd})$
şi Q **sat** $T(x,ur,x^{\surd})$
şi P nu diverge
atunci $(P;Q)$ **sat** $(\exists y,s,t.\ ur=(s;t)\wedge S(x,s,y)\wedge T(y,t,x^{\surd}))$

În această lege, x este o listă de variabile din alfabetele lui P şi Q, $x^{\surd}$ este o listă a variantelor cu $\surd$ şi y o listă cu acelaşi număr de variabile noi.

Specificarea condiţiei este aceeaşi ca pentru prima componentă dacă condiţia este adevărată şi ca pentru a doua dacă este falsă

L4 Dacă P **sat** S şi Q **sat** T
atunci $(P \lhd b \rhd Q)$ **sat** $((b \wedge S) \vee (\neg b \wedge T))$

O altă formă a acestei legi este câte o dată mai convenabilă

L4A Dacă P **sat** $(b \Rightarrow S)$ şi Q **sat** $(\neg b \Rightarrow S)$
atunci $(P \lhd b \rhd Q)$ **sat** S

Exemplu

X10 Fie $COND=(q:=q+1;\ r:=r-y;\ X) \lhd r \geq y \rhd SKIP$
şi X **sat** $(T(n) \Rightarrow BUCLĂDIV)$
atunci $COND$ **sat** $(T(n+1) \Rightarrow BUCLĂDIV)$

Cele două condiţii suficiente pentru această concluzie s-au demonstrat în X7 şi X9, iar rezultatul decurge din L4A. □

Demonstrarea specificării unei bucle foloseşte definiţia recursivă dată în 5.5 D1 şi în legea pentru recursivitatea fără gardă (3.7.1 L8). Dacă R este specificaţia dorită a buclei trebuie să găsim o specificare $S(n)$ astfel încât $S(0)$ să fie totdeauna adevărată şi de asemenea

$$(\forall n.S(n)) \Rightarrow R$$

O metodă generală pentru a construi $S(n)$ este de a găsi un predicat $T(n,x)$ care descrie condiţiile din starea iniţială x astfel încât bucla să se termine cu siguranţă în mai puţin de n paşi. Deci definim

$$S(n)=(T(n,x) \Rightarrow R)$$

În mod clar, nici o buclă nu poate avea 0 paşi astfel că dacă $T(n,x)$ a fost corect definită, $T(0,x)$ este falsă şi deci $S(0)$ va fi adevărată. Rezultatul demonstraţiei asupra specificării buclei va fi $\forall n.S(n)$ sau

$$\forall n.(T(n,x) \Rightarrow R$$

Deoarece n este o variabilă astfel aleasă ca să nu apară în R, aceasta este echivalent cu

$\exists n.(T(n,x) \Rightarrow R$

Nu există nici o specificare mai tare decât $\exists n\, T(n,x)$ ca precondiţie astfel încât bucla să se termine într-un număr finit de paşi.

În fine, trebuie să dovedim că corpul buclei satisface specificaţia. Deoarece ecuaţia recursivă pentru o buclă implică o condiţie, această activitate se desparte în două. De aceea avem legea generală

L5 Dacă $\neg T(0,x)$ şi $T(n,x) \Rightarrow b$
şi $SKIP$ **sat** $(\neg b \Rightarrow (T(n,x) \Rightarrow R))$
şi $(X$ **sat** $(T(n,x) \Rightarrow R)) \Rightarrow ((Q;X)$ **sat** $(b \Rightarrow (T(n+1,x) \Rightarrow R)))$
atunci $(b * Q)$ **sat** $((\exists n.T(n,x)) \Rightarrow R)$

Exemplu

X11 Dorim să dovedim că programul pentru metoda lungă de împărţire prin diferenţă repetată (5.5 X5) satisface specificaţia lui DIV. Activitatea se desparte natural în două. A doua parte, mai dificilă de arătat, este că bucla satisface specificaţia corespunzător formulată şi anume

$(r \geq y) * (q := q+1;\ r := r-y)$ **sat** $(y > 0 \Rightarrow BUCLĂDIV)$

Mai întâi e necesar să formulăm condiţia astfel ca bucla să se termine în mai puţin de n iteraţii

$T(n) = r < n \times y$

Aici $T(0)$ este evident fals. Clauza

$\exists n.T(n)$

este echivalentă cu

$y > 0$

care este precondiţia ca bucla să se termine. Restul paşilor pentru demonstrarea buclei au fost făcuţi în X7 şi X5. În continuare demonstraţia este un simplu exerciţiu. □

Legile date în acest paragraf sunt destinate unui calcul al corectitudinii totale pentru programele pur secvenţiale care nu conţin intrări sau ieşiri. Dacă Q este un astfel de program, atunci o demonstraţie a faptului că

Q **sat** $(P(x) \wedge ur \neq \langle\rangle \Rightarrow ur = \langle\surd\rangle \wedge R(x, x^{\surd}))$ (1)

stabilește că dacă $P(x)$ este adevărată pentru valorile inițiale ale variabilelor când Q este activ atunci Q se va termina și $R(x,x^{\surd})$ va descrie relația dintre valorile inițiale ale lui x și valorile finale $x^{\surd}$. Astfel $(P(x),R(x,x^{\surd}))$ este o pereche precondiție/postcondiție în sensul lui Cliff Jones. Dacă $R(x^{\surd})$ nu specifică valoarea inițială x, (1) este echivalentă cu

$$P(x)\Rightarrow wp(Q,R(x))$$

unde *wp* este precondiția cea mai slabă a lui Dijkstra.

Astfel în capul special al programelor necomunicante, metodele de demonstrare sunt echivalente matematic cu acelea deja familiare, cu toate că menționarea explicită a lui "*ur*=<>" și "*ur*=<√>" le face notațional mai stângace. Acèst lucru suplimentar este desigur necesar și de aceea acceptabil când metodele sunt extinse la procese secvențiale comunicante.

5.5.3 Implementare

Stările inițiale și finale ale proceselor secvențiale pot fi reprezentate ca o funcție care face să-i corespundă fiecărui nume de variabilă valoarea sa. Un proces secvențial este definit ca o funcție care face să-i corespundă stării sale inițiale comportarea sa ulterioară. Terminarea cu succes este reprezentată de atomul "*SUCCES*. Un proces care este gata de a se termina va accepta acest simbol căruia îi va corespunde nu un alt proces ci starea finală a variabilelor sale. Astfel procesul *SKIP* are ca parametru o stare inițială, acceptă atomul "*SUCCES* ca singura acțiune și livrează starea inițială ca stare finală

SKIP=*λs.λy.* **if** *y*≠"*SUCCES* **then** "*BLIP* **else** *s*

O asignare este similară cu *SKIP*, cu deosebirea că starea inițială este puțin diferită

asign(*x*,*e*)=*λs.λy.* **if** *y*≠"*SUCCES* **then** "*BLIP*
else *actualizează* (*s*,*x*,*e*)

unde *actualizează*(*s*,*x*,*e*)=*λy.* **if** *y*=*x* **then** *eval*(*e*,*s*) **else** *s*(*y*)
și *eval*(*e*,*s*) este rezultatul evaluării expresiei *e* în starea *s*.

Dacă *e* este nedefinită în starea *s* nu contează ce se întâmplă. Aici, pentru simplitate, am implementat numai o asignare simplă. Asignarea multiplă este puțin mai complicată.

Pentru a implementa compunerea secvențială este necesar să testăm mai întâi dacă primul operand s-a terminat cu succes. Dacă da, starea sa finală este trecută la al doilea operand. Dacă nu, prima acțiune este aceea a primului operand

secvență(*P*,*Q*)=

λ*s*. **if** *P*(*s*) ("*SUCCES*)≠"*BLIP* **then** *Q*(*P*(*s*) ("*SUCCES*))
else λ*v*. **if** *P*(*s*)(*v*)="*BLIP* **then** "*BLIP*
else *secvență*(*P*(*s*)(*v*),*Q*)

Implementarea condiției se face printr-o condiție

condiție(*P*,*b*,*Q*)=λ*s*. **if** *eval*(*b*,*s*) **then** *P*(*s*) **else** *Q*(*s*)

Implementarea buclei ($b*Q$) este lăsată ca exercițiu.

De notat că definiția lui *secvență* dată mai sus este mai complicată decât aceea dată în paragraful 5.3.3, din cauză că are o stare *s* ca prim argument și trebuie să transfere această stare ca prim argument operanzilor. Din nefericire, o complexitate asemănătoare trebuie introdusă în definițiile tuturor celorlalți operatori dați în capitolele anterioare. O soluție mai simplă ar fi să modelăm variabilele ca procese subordonate. Dar aceasta ar fi probabil un lucru mai complicat și mai puțin eficient decât utilizarea memorării în acces aleator convențională. Când considerațiile de eficiență sunt adăugate celor matematice, sunt motive întemeiate pentru introducerea variabilelor de program asignabile ca noi concepte decât definirea lor prin conceptele deja introduse.

6 Resurse partajate

6.1 Introducere

În paragraful 4.5 am introdus conceptul de proces subordonat notat (*m*:*R*), a cărui singură funcţiune este îndeplinirea serviciilor faţă de un singur proces principal *S*. Pentru aceasta am folosit notaţia

$(m:R//S)$

Presupunem acum că *S* conţine sau constă din două procese concurente ($P||Q$) şi ambele procese *P* şi *Q* necesită servicii ale aceluiaşi proces subordonat (*m*:*R*). Din nefericire, nu este posibil ca *P* şi *Q* să comunice amândouă cu (*m*:*R*) de-a lungul aceloraşi canale, din cauză că aceste canale ar trebui să fie atât în alfabetul lui *P* cât şi al lui *Q* şi atunci definiţia operatorului || ar impune pentru comunicaţiile lui *P* şi *Q* cu (*m*:*R*) transmiterea numai a aceloraşi mesaje simultan – care (cum s-a explicat şi în 4.5 X6) este departe de efectul dorit. Ceea ce dorim este o posibilitate de întreţesere a comunicaţiilor între *P* şi (*m*:*R*) cu acelea între *Q* şi (*m*:*R*). În acest mod (*m*:*R*) se comportă ca o resursă partajată lui *P* şi *Q*. Fiecare din procese o va putea folosi independent şi interacţiunile lor cu ea vor fi întreţesute.

Când identitatea tuturor proceselor care partajează o resursă este cunoscută dinainte este posibil de aranjat ca fiecare proces partajant să folosească mulţimi de canale diferite pentru a comunica cu resursa partajată. Această tehnică s-a utilizat în problema mesei celor cinci filozofi (paragraful 2.5). Fiecare furculiţă era partajată de doi filozofi vecini şi valetul era partajat între toţi cinci. Alt exemplu a fost 4.5 X6, în care un buffer era partajat între două procese, unul dintre ele utilizând numai canalul din stânga, celălalt numai canalul din dreapta. O metodă generală de partajare este realizată prin etichetare multiplă (paragraful 2.6.4) care efectiv creează destule canale separate pentru comunicaţia cu fiecare proces partajant. Comunicaţiile individuale de-a lungul acestor canale sunt arbitrar întreţesute. Dar această metodă necesită ca numele tuturor proceselor partajante să fie cunoscut dinainte şi de aceea nu este adecvată pentru un proces subordonat care se intenţionează să deservească nevoile unui proces principal ce la rândul lui se compune dintr-un număr de

sub-procese concurente. Acest capitol introduce tehnici pentru partajarea resurselor între mai multe procese, chiar când numărul şi identitatea lor nu sunt cunoscute dinainte. Se vor ilustra tehnicile prin exemple din proiectarea unui sistem de operare.

6.2 Partajarea prin întreţesere

Problema evidenţiată în paragraful 6.1 provine din utilizarea operatorului || pentru descrierea comportării concurente a proceselor; problemă ce poate fi adeseori evitată utilizând în locul operatorului || forma întreţesută de concurenţă (*P*|||*Q*). Aici, *P* şi *Q* au acelaşi alfabet şi comunicaţiilor lor cu procese externe (partajate) este arbitrar întreţesută. Desigur, se interzice directa comunicare între *P* şi *Q*, dar o comunicare indirectă poate fi stabilită cu ajutorul unui proces subordonat partajat aşa cum se arată în 4.5 X6 şi X2 de mai jos.

Exemple

X1 Subrutina partajată

dub:*DUBLU*//(*P*|||*Q*)

Aici, atât *P* cât şi *Q* pot conţine apeluri ale procesului subordonat

(*dub*.*stânga*!v→*dub*.*dreapta*?*x*→*SKIP*)

Chiar când aceste perechi de comunicaţii de la *P* şi *Q* sunt arbitrar întreţesute, nu există pericolul ca unul din procese să obţină accidental un răspuns care ar fi trebuit să fie primit de celălalt. Pentru a ne asigura de aceasta, toate subprocesele procesului principal trebuie să respecte o strictă alternanţă a comunicaţiilor pe canalul din stânga cu comunicaţiile pe cel din dreapta corespunzătoare procesului subordonat partajat. Din acest motiv, pare rezonabil să introducem o notaţie specializată a cărei utilizare exclusivă va garanta aplicarea disciplinei cerute. Notaţia în cauză este o reminescenţă a unui apel procedural dintr-un limbaj de nivel înalt, cu deosebirea că parametrii valoare transmişi sunt precedaţi de ! iar parametrii rezultat de ?, astfel că

dub!*x*?*y*=(*dub*.*stânga*!*x*→*dub*.*dreapta*?*y*→*SKIP*) □

Efectul dorit de partajare prin întreţesere este ilustrat prin următoarele serii de transformări algebrice. Când două procese partajate încearcă simultan să folosească subrutina partajată, perechile corespunzătoare de comunicaţii

sunt luate în ordine arbitrară, dar componentele unei perechi de comunicaţii cu un proces nu sunt niciodată separate de comunicaţia cu celălalt. Pentru uşurinţă folosim următoarele abrevieri

$d!v$ pentru $d.stânga!v$
$d?v$ pentru $d.dreapta?v$ } în procesul principal

$!v$ pentru $dreapta!v$
$?v$ pentru $stânga?v$ } în procesul subordonat

Fie $D = ?x \to !(x+x) \to D$
şi $P = d!3 \to d?y \to P(y)$
şi $Q = d!4 \to d?z \to Q(z)$
şi $R = (d:D // (P ||| Q))$ ca în X1 de mai sus

Atunci $P ||| Q = (d!3 \to ((d?y \to P(y)) ||| Q)$
$\square d!4 \to (P ||| (d?z \to Q(z))))$ din 3.6.1 L7

Fiecare din procesele partajante începe cu emisia către procesul partajat. Procesului partajat i se oferă şansa de a alege între cele două procese principale. Dar procesul partajat doreşte să le accepte pe oricare astfel că după mascarea alegerii devine nedeterminist

$$(d:D // (P ||| Q)) = ((d:(!3+3 \to D)) // ((d?y \to P(y)) ||| Q))$$
$$\sqcap ((d:(!4+4 \to D)) || (P ||| (d?z \to Q(z))))$$
$$= (d:D // (P(6) ||| Q)) \sqcap (d:D // (P ||| Q(8)))$$

{ 4.5.1.L1, 3.5.1.L5, etc. }

Procesul partajat oferă rezultatul său indiferent căruia din procesele partajante este gata să-l accepte. Deoarece cel puţin unul din ele stă în recepţie, acesta va fi procesul ce a asigurat argumentul care genererează rezultatul. De aceea este aşa de importantă stricta alternare a transmisiei cu recepţia în apelarea unei subrutine partajate.

X2 Structură de date partajate

Într-un sistem de rezervare a biletelor de avion rezervările de locuri se fac de către mulţi funcţionari a căror acţiuni sunt întreţesute. Fiecare rezervare adaugă un pasager listei de zbor şi returnează un răspuns dacă pasagerul a fost acceptat sau nu. În acest exemplu foarte simplu, mulţimea implementată în 4.5.X8 va servi ca proces subordonat partajat cu numele de la indicativul zborului

*RO*109:*SET*//(... (*FUNC*|||*FUNC*|||...)...)

Fiecare *FUNC* înregistrează un pasager prin apelul

*RO*109!*nr_pas*?*x*

care corespunde la

(*RO*109.*stânga*!*nr_pas*→*RO*109.*dreapta*?*x*→*SKIP*) □

În aceste două exemple, fiecare ocazie de a utiliza resursa partajată implică exact două comunicații, una pentru a transmite parametrii şi alta pentru a recepționa rezultate. După fiecare pereche de comunicații, procesul subordonat trece din nou în starea în care este gata să deservească alt proces sau pe acelaşi din nou. Dar frecvent vrem să ne asigurăm că o întreagă serie de comunicații are loc între două procese fără pericolul de a interfera cu un al treilea. De exemplu, un dispozitiv scump de transmisie ar putea fi partajat de mai multe procese concurente. De fiecare dată, un număr de linii constituind un fişier trebuie transmise consecutiv fără pericolul întrețeserii liniilor trimise de alt proces. Pentru aceasta, transmisia fiecărui fişier trebuie precedată de un eveniment *ocupă* care obține utilizarea exclusivă a resursei, iar la sfârşit, resursa trebuie disponibilizată din nou prin evenimentul *eliberează*.

X3 Imprimantă partajată

LP=*ocupă*→μ*X*.(*stânga*?*s*→*h*!*s*→*X* | *eliberează*→*LP*)

Aici, *h* este canalul care face legătura între procesul *LP* şi hardware-ul imprimantei. După ocuparea imprimantei, procesul copie linii succesive de la canalul său stâng către canalul hardware până când un semnal de eliberare îl readuce în starea sa inițială în care este disponibil pentru alte procese. Acest proces este utilizat ca o resursă partajată

lp:*LP*// ... (*P*|||*Q*) ...

În interiorul lui *P* şi *Q* transmisia unor linii dintr-un fişier este încadrată de evenimentele *lp.ocupă* şi *lp.eliberează*

lp.ocupă→...*lp.stânga*!"*C.IONESCU*"→...
lp.stânga!*linia_următoare*→...*lp.eliberează*→... □

X4 O îmbunătățire a lui X3

Când o imprimantă este partajată între mai mulţi utilizatori, zona hârtiei conţinând fiecare fişier listat trebuie detaşată manual dintre eventualele fişiere anterioare şi următoare la sfârşitul fiecărei tipăriri conform cu X3. De aceea, de obicei, hârtia de imprimantă este împărţită în pagini care sunt separate de perforaţii şi hardware-ul imprimantei permite operaţia *avans* care mişcă hârtia rapid către sfârşitul paginii curente – sau mai bine zis la începutul unei pagini noi. Pentru a ajuta separarea, fişierele trebuie cadrate în pagină şi trebuie imprimat un rând complet de asteriscuri la sfârşitul ultimei pagini a fişierului precum şi la începutul primei pagini. Pentru a evita confuziile nu se permite imprimarea unei linii complete de "*" în mijlocul fişierului.

$$LP=(h!avans\to h!asteriscuri\to ocup\breve{a}\to h!asteriscuri\to$$
$$\mu X.(st\hat{a}nga?s\to \textbf{if } s\neq asteriscuri \textbf{ then } (h!s\to X) \textbf{ else } X$$
$$|\ elibereaz\breve{a}\to LP)$$

Această versiune de *LP* are aceeaşi funcţionalitate ca cea anterioară. □

În ultimele două exemple, utilizarea evenimentelor *ocupă* şi *eliberează* evită întreţeserea arbitrară de linii din fişiere distincte şi aceasta fără pericolul de blocaj. Dar dacă mai mult de o resursă este partajată în această manieră, riscul blocajului nu poate fi ignorat.

X5 Blocaj

Ann şi Mary sunt bune prietene şi bucătărese. Ele folosesc în comun o cratiţă şi o tigaie pe care le solicită, utilizează şi eliberează după cum au nevoie

$$VASE=(ocup\breve{a}\to utilizeaz\breve{a}\to utilizeaz\breve{a}\to\ldots\to elibereaz\breve{a}\to VASE)$$
$$cratit\breve{a}:VASE//tigaie:VASE//(ANN||MARY)$$

Ann găteşte potrivit unei reţete care cere întâi cratiţa şi apoi tigaia, în timp ce Mary are nevoie întâi de tigaie şi apoi de cratiţă

$$ANN=\ldots cratit\breve{a}.ocup\breve{a}\to\ldots\to tigaie.ocup\breve{a}\to\ldots$$
$$MARY=\ldots tigaie.ocup\breve{a}\to\ldots\to cratit\breve{a}.ocup\breve{a}\to\ldots$$

Din nefericire, gospodinele decid să înceapă aproximativ în acelaşi timp. Fiecare solicită primul vas corespunzător, dar când au nevoie de al doilea îşi dau seama că nu îl pot avea deoarece este utilizat de prietenă.

Povestea lui Ann şi Mary poate fi ilustrată grafic (fig. 6.1) unde timpul lui Ann este pe axa verticală iar a lui Mary pe cea orizontală. Sistemul porneşte din colţul din stânga jos, originea timpului pentru ambele gospodine. De fiecare dată când Ann face o acţiune, sistemul se mişcă în sus un pas. De fiecare dată când Mary face o acţiune, sistemul se mişcă la dreapta un pas.

Traiectoria urmată arată clar întreţeserea acţiunilor lui Ann şi Mary. Din fericire, această traiectorie ajunge în colţul din dreapta sus unde ambele gospodine se bucură că masa este gata.

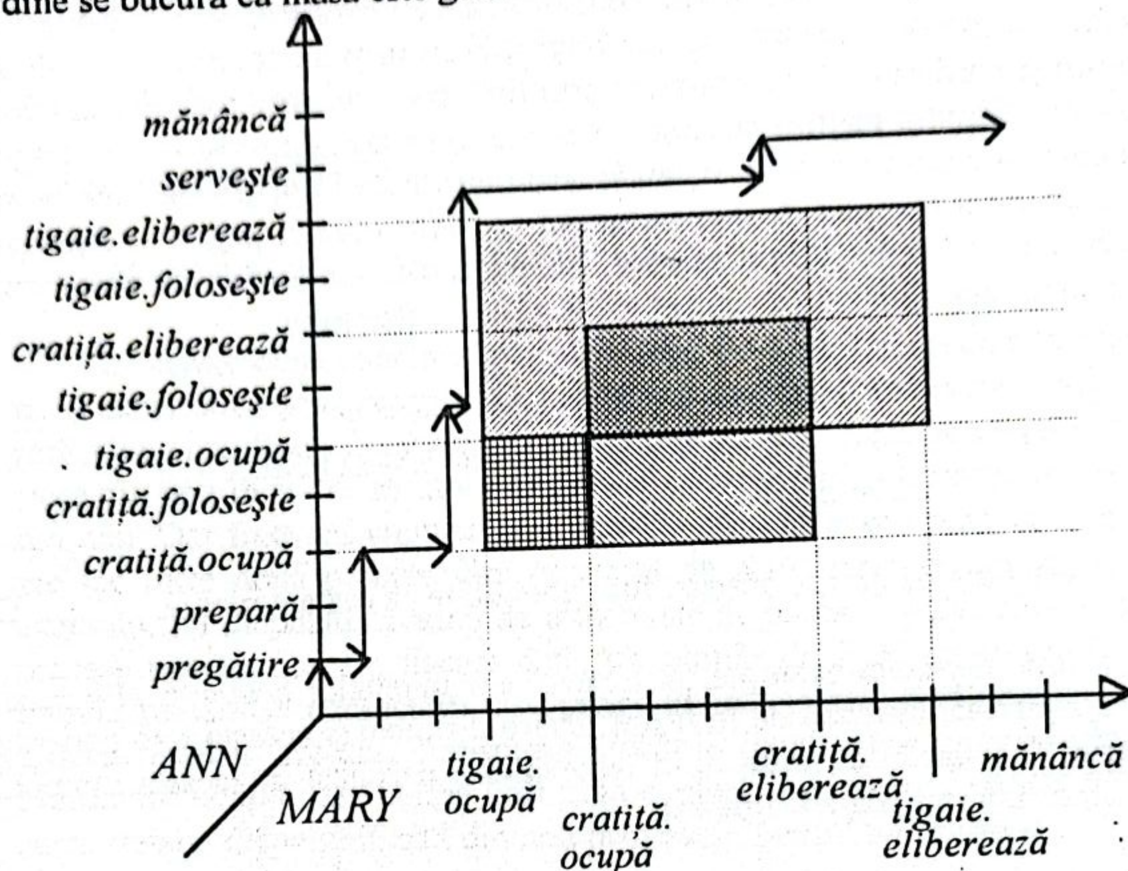


Figura 6.1

Dar acest sfârşit fericit este nesigur. Deoarece nu pot folosi simultan un vas partajat există anumite regiuni în spaţiul stărilor prin care traiectoria nu poate trece. De exemplu, în regiunea haşurată cu ▨ ambele gospodine ar folosi tigaia şi aceasta nu este posibil. Analog, excluderea în utilizarea simultană a cratiţei interzice intrarea în regiunea haşurată cu ▧. Astfel dacă traiectoria atinge marginea uneia din zonele interzise poate numai să urmeze marginea (vertical sau orizontal). În acest timp, una din gospodine aşteaptă eliberarea vasului de către cealaltă.

Să considerăm acum zona cu ▦. Dacă traiectoria intră vreodată în această zonă va sfârşi inevitabil în blocaj în colţul din dreapta sus al zonei. Scopul desenului este de a arăta că pericolul blocajului provine numai din zona interzisă din întreaga regiune mărginită : celelalte zone sunt sigure. Singura modalitate de a preveni blocajul este de a extinde regiunea interzisă şi în zonele periculoase. O tehnică ar fi să introducem o resursă artificială în plus care ar fi solicitată înaintea oricărui vas şi nu trebuie eliberată până ce ambele vase nu s-au eliberat. Această soluţie este similară celei găsite prin introducerea valetului în povestea mesei filozofilor (paragraful 2.5.3.), unde permisiunea de a sta jos este un fel de resursă din care numai patru copii sunt par-

tajate de cei cinci filozofi. O soluţie mai uşoară este de a insista ca gospodina care vrea ambele vase să solicite mai întâi tigaia. Acest exemplu este datorat lui E.W. Dijkstra. □

Soluţia cea mai uşoară sugerată pentru exemplul anterior se poate generaliza la orice număr de utilizatori şi resurse. În condiţiile în care este o ordine în care utilizatorii solicită resursele nu există riscul de blocaj. Utilizatorii ar trebui să-şi elibereze resursele cât mai repede posibil după ce le-au folosit, ordinea de eliberare necontând. Utilizatorii pot solicita resurse şi într-o ordine oarecare, în condiţiile în care în momentul solicitării toate resursele sunt deja eliberate fiind ordonate apoi corespunzător. Respectarea acestei discipline a solicitării şi eliberării resurselor poate fi verificată adesea printr-o vizualizare a textului proceselor utilizator.

6.3 Partajarea memoriei

Obiectivul acestui paragraf este de a aduce argumente împotriva utilizării memoriei partajate. Paragraful poate fi sărit de cei care sunt deja convinşi de acest lucru.

Comportarea sistemelor de procese concurente poate fi uşor implementată pe un singur computer convenţional printr-o tehnică numită *partajarea timpului* (*timesharing*), în care un singur procesor execută pe rând fiecare din procese, cu schimbarea procesului prin intermediul unei întreruperi de la un dispozitiv extern sau de la un ceas sistem. În această implementare este foarte uşor să permitem proceselor concurente să partajeze locaţiile de memorie comună care sunt accesate şi asignate simplu cu ajutorul instrucţiunilor maşină uzuale din programul corespunzător fiecarui proces.

O locaţie de memorie partajată poate fi modelată în teoria noastră ca o variabilă partajată (4.2 X7) cu un nume simbolic adecvat, de exemplu

$$(contor{:}VAR /\!/ (contor.stânga!0 \rightarrow (P|||Q)))$$

Memorarea partajată poate fi uşor deosebită de o memorare locală descrisă în 5.5. Simplitatea legilor pentru formalismul proceselor secvenţiale derivă numai din faptul că variabila este actualizată de cel mult un proces şi aceste legi nu tratează multiplele pericole care provin din întreţeserea arbitrară a asignărilor în diferite procese.

Aceste pericole sunt mai clar ilustrate de următorul exemplu.

X1 Interferenţa

Variabila partajată *contor* este utilizată pentru a ţine socoteala numărului total de apariţii ale unui eveniment important. La fiecare apariţie a evenimentului,

procesul relevant *P* sau *Q* încearcă să actualizeze *contor* prin perechea de comunicații

contor.dreapta?x; contor.stânga!(x+1)

Din nefericire aceste două evenimente de comunicații pot fi întrețesute cu o pereche identică de evenimente de comunicații de la un alt proces, rezultând secvența

contor.dreapta?x→contor.dreapta?y→contor.stânga!(y+1)→
contor.stânga!(x+1)→...

Ca o consecință, valoarea lui *contor* este incrementată numai de unul din procese, în loc de două. Acest fel de eroare este cunoscut ca interferență și este o eroare frecventă în proiectarea proceselor care partajează memorie comună. Mai mult, apariția erorii este complet nedeterministă. Ea nu este reproductibilă sigur și de aceea este imposibil (aproape) să diagnozăm eroarea prin tehnici de testare convenționale. Ca un principal rezultat, eu suspectez că există câteva sisteme de operare de largă utilizare care regulat produc ușoare inexactități la statistici și contabilizări. □

O posibilă soluție a acestei probleme este să ne asigurăm că nici o schimbare a procesului nu are loc în timpul unei secvențe de acțiuni, ceea ce înseamnă o protecție la întrețesere. O astfel de secvență este cunoscută ca *regiune critică*. La implementarea pe un singur procesor, excluderea necesară este adesea realizată prin inhibarea tuturor întreruperilor pe perioada regiunii critice. Soluția are efectul nedorit al întârzierii răspunsului la întreruperi și, mai rău, se prăbușește când este adăugată o a doua unitate de procesare în computer.

O soluție mai bună a fost sugerată de E.W. Dijkstra în introducerea sa la semafoarele de excludere binare. Un semafor poate fi descris ca un proces care se angajează succesiv în acțiunile *P* și *V*

SEM=(P→V→SEM)

Aceasta este declarată ca o resursă partajată

(mutex:SEM//...)

Fiecare proces, la intrarea în regiunea critică, trebuie să trimită semnalul

mutex.P

și la ieșirea din regiunea critică trebuie să se angajeze în evenimentul

mutex.V

Astfel regiunea critică în care *contor* este incrementată ar apărea

mutex.P→
contor.dreapta?x→*contor.stânga*!(*x*+1)→
mutex.V→...

În condiţiile în care toate procesele respectă această disciplină este imposibil ca două procese să interfere între ele actualizând *contor*. Dar dacă unul din procese omite *P* sau *V* sau le ia în ordine greşită efectul va fi haotic şi se va risca o eroare subtilă, dezastruoasă (sau mai rău).

O cale mai sigură de a evita interferenţa este de a clădi protecţia necesară prin însăşi modelarea memoriei partajate, profitând de faptul că ştim cum o vom utiliza. De exemplu, dacă o variabilă trebuie folosită numai pentru socoteli, atunci operaţia care o incrementează ar putea fi o operaţie atomică

contor.sus

şi resursa partajată ar putea fi proiectată ca MM_0 (1.1.4 X2)

contor:MM_0//(...*P*|||*Q*...)

De fapt sunt destule motive pentru a recomanda ca fiecare resursă partajată să fie proiectată special pentru scopul ei iar în proiectarea unui sistem folosind concurenţa memoria principală nu trebuie partajată. Aceasta nu numai că evită pericolul interferenţei accidentale, ci de asemenea, permite realizarea unui proiect care poate fi implementat eficient pe reţele cu elemente de procesare distribuite ca de altfel şi pe un singur procesor sau multiprocesor cu memorie partajată fizic.

6.4 Resurse multiple

În paragraful 6.2 am descris cum un număr de procese concurente cu comportare diferită ar putea partaja un singur proces subordonat. Fiecare proces partajant respectă o disciplină a alternării transmisiei cu recepţia sau alternării semnalelor de ocupare si eliberare, pentru a se asigura că în orice moment resursa este utilizată de cel mult unul din potenţialele procese partajante. Astfel de resurse sunt cunoscute ca *reutilizabile serial*. În acest paragraf introducem masive de procese pentru a reprezenta o resursă multiplă cu comportare identică. Un indice în masiv ne asigură că fiecare element comunică în siguranţă cu procesul care l-a solicitat.

De aceea vom utiliza pe scară largă indici şi operatori indexaţi cu semnificaţie evidentă. De exemplu

$$\underset{i<12}{\|} P_i = (P_0 \| P_1 \| \ldots \| P_{11})$$

$$\underset{i<4}{|||} P = (P ||| P ||| P ||| P)$$

$$\underset{i\geq 0}{\|} P_i = (P_0 \| P_1 \| \ldots)$$

$$\underset{i\geq 0}{\square} (f(i) \rightarrow P_i) = (f(0) \rightarrow P_0 | f(1) \rightarrow P_1 | \ldots)$$

În ultimul exemplu se impune ca f să fie injectivă, astfel încât alegerea între alternative să fie făcută numai de mediu.

Exemple

X1 Subrutină reentrantă

O subrutină partajată care este folosită serial poate fi utilizată numai de un singur proces apelant odată. Dacă execuţia subrutinei necesită calcule considerabile, ar putea fi întârzieri semnificative în procesele apelante. Dacă sunt disponibile mai multe procesoare, există suficiente motive de a permite unor copii ale subrutinei să se lanseze concurent pe diferite procesoare. O subrutină capabilă de mai multe copii concurente este cunoscută ca *reentrantă* şi este definită ca un masiv de procese concurente

$$dub{:}(\underset{i<27}{\|}(i{:}DUBLU))/\!\!/\ldots$$

Un apel tipic al acestei subrutine ar putea fi

$$dub.3.stânga!30 \rightarrow dub.3.dreapta?y \rightarrow SKIP)$$

Utilizarea indexului 3 ne asigură că rezultatul apelului este obţinut de la aceeaşi instanţă a lui *dub* căreia i-au fost trimise argumentele, chiar dacă alt proces concurent ar putea apela în acelaşi timp altă instanţă din masiv, rezultând o întreţesere a mesajelor

*dub.3.stânga.*30,...*dub.2.stânga.*20,...*dub.3.dreapta.*60,...*dub.2.dreapta.*40,...

Când un proces apelează o subrutină reentrantă nu contează care element din masiv va răspunde la apel, oricare va fi liber va fi la fel de bun. Din acest

motiv, decât să specificăm un index particular, de exemplu 2 sau 3, un proces particular ar putea lăsa ca selecția să se facă arbitrar prin construcția

$$\underset{i \geq 0}{\square}(dub.i.stânga!30 \rightarrow dub.i.dreapta?y \rightarrow SKIP)$$

Se observă respectarea disciplinei esențiale ca același index să fie folosit pentru trimiterea argumentelor și (imediat după) pentru recepționarea rezultatului. □

În exemplul arătat mai sus există o limită arbitară de 27 de activări simultane ale subrutinei. Deoarece este relativ echitabil de aranjat ca un singur procesor să-și împartă atenția între un număr mare de procese, astfel de limite arbitrare pot fi evitate prin introducerea unui masiv infinit de procese concurente

$$dub:(\underset{i \geq 0}{\|} i:D)$$

unde D poate fi proiectat să servească numai un singur apel și apoi să se oprească

$$D = stânga?x \rightarrow dreapta!(x+x) \rightarrow STOP$$

O subrutină fără limite în reentranța ei este cunoscută ca *procedură*.
Intenția utilizării unei proceduri este ca efectul fiecărui apel

$$\underset{i \geq 0}{\square}(dub.i.stânga!30 \rightarrow dub.i.dreapta?y \rightarrow SKIP)$$

să fie identic cu apelul unui proces subordonat D declarat adiacent imediat apelului

$$(dub:D // (dub.stânga!30 \rightarrow dub.dreapta?y \rightarrow SKIP))$$

Această tehnică este cunoscută ca un apel *local* de procedură, deoarece sugerează execuția procedurii pe același procesor ca și procesul apelant, în timp ce apelul unei proceduri partajate este cunoscut ca apel *la distanță*, deoarece sugerează execuția pe un posibil procesor aflat la distanță. Deoarece efectul unor apeluri locale sau la distanță se dorește a fi același, motivele pentru utilizarea apelului la distanță pot fi numai politice sau economice – de exemplu ținerea codului procedurii secret sau lansarea ei pe o mașină cu fa-

cilități speciale care sunt prea scumpe de asigurat pe mașina pe care se rulează procesele utilizator.

Un exemplu tipic de facilitate scumpă este un suport de înmagazinare de mare volum, ca de exemplu un disc sau memoria cu bule.

X2 Memorie partajată secundară

Un mediu de memorare este împărțit în *S* sectoare care pot fi citite și scrise independent. Fiecare sector poate reține o cantitate de informație pe care-o recepționează prin canalul din stânga și o emite prin canalul din dreapta. Din nefericire, mediul de memorare este implementat într-o tehnologie cu citit distructiv, astfel că fiecare bloc scris poate fi citit o singură dată. Astfel, fiecare sector se comportă mai degrabă ca *COPIE* (4.2 X1) decât ca *VAR* (4.2 X7). Întregul mediu de memorare secundară este un masiv de astfel de sectoare, indexat după numere mai mici ca *S*

$$SMEM = \|_{i<S}\, i{:}COPIE$$

Această memorie se dorește a fi utilizată ca proces subordonat

$(mems{:}SMEM /\!/ \ldots)$

În procesul principal, memoria poate fi folosită prin comunicații

$mems.i.stânga!sec \rightarrow \ldots mems.i.dreapta?y \rightarrow \ldots$

Memoria secundară poate fi de asemenea partajată de procese concurente. În acest caz acțiunea

$$\square_{i<S}(mems.i.stânga!sec \rightarrow \ldots)$$

va solicita simultan un sector liber arbitrar cu numărul *i* și va scrie valoarea *sec* în el. Similar, *mem.i.dreapta*?*x* va face într-o singură acțiune citirea conținutului sectorului *i* în *x* și eliberarea acestui sector pentru altă folosire cu altă ocazie, foarte posibil de un alt proces. Această simplificare este motivul real pentru utilizarea lui *COPIE* ca model de comportare pentru fiecare sector. Povestea cu citirea distructivă este doar o poveste. □

Desigur, partajarea cu succes a memoriei secundare necesită o disciplină extremă din partea proceselor partajante. Un proces poate recepționa de la un sector numai dacă același proces a emis de curând către același sector și fiecare emisie trebuie eventual urmată de o recepție. Nerespectarea unei astfel de discipline va conduce la blocaj sau mai rău la confuzie. Metode de întărire a

acestei discipline fără complicaţii vor fi introduse după următorul exemplu şi vor fi complet ilustrate în proiectarea ulterioară de module ale unui sistem de operare (paragraful 6.5)

X3 Două imprimante

Două imprimante identice sunt disponibile pentru a servisa cererile unei colecţii de procese utilizator. Ambele necesită protecţie la întreţesere care a fost asigurată de *LP* (6.2 X4). De aceea declarăm un masiv de două instanţe a lui *LP* fiecare indexat de un număr natural indicând poziţia în masiv.

*LP*2=(0:*LP*||1:*LP*)

Acestui masiv îi poate fi dat un nume pentru utilizarea ca resursă partajată

(*lp*:*LP*2||...)

Fiecare instanţă a lui *LP* este acum prefixată de două ori, odată prin nume, odată prin indice. Astfel, comunicaţiile cu procesul utilizator au trei sau patru componente, de exemplu

lp.0.*ocupă*,*lp*.1.*stânga*."*C.IONESCU*",...

Ca în cazul unei proceduri reentrante, când un proces are nevoie să solicite una din resursele identice dintr-un masiv, într-adevăr nu contează care element al masivului va fi selectat cu o anumită ocazie. Orice element care este gata să răspundă la semnalul de solicitare va fi acceptabil. O formă generală de alegere va face selecţia arbitrară cerută

$$\square_{i\geq 0}(lp.i.ocupă\rightarrow\ldots\ lp.i.stânga!x\rightarrow\ldots\ lp.i.eliberează\rightarrow SKIP)$$

Aici, acţiunea iniţială *lp.i.ocupă* va solicita oricare dintre cele două procese *LP* care este gata pentru acest eveniment. Dacă nici unul nu-i gata, procesul solicitant va aştepta. Dacă amândouă sunt gata, alegerea între ele este nedeterministă. După solicitarea iniţială variabila mărginită *i* ia ca valoare indexul resursei selectate şi toate comunicaţiile ulterioare vor fi corect direcţionate către această resursă. □

Când o resursă partajată a fost solicitată pentru o folosire temporară într-un proces, resursa se doreşte a se comporta exact ca un proces subordonat declarat local, comunicând numai cu procesul său utilizator.

De aceea vom adapta şi rescrie familiara notaţie pentru subordonare

(*fispropriu*::*lp*//... *fispropriu.stânga*!*x*...)

în loc de mai complexa construcţie

$$\underset{i\geq 0}{\square}(lp.i.ocup\breve{a}\rightarrow \ldots\ lp.i.st\hat{a}nga!x\rightarrow \ldots;\ lp.i.elibereaz\breve{a}\rightarrow SKIP)$$

Aici, numele local *fispropriu* a fost introdus pentru a desemna numele indexat *lp.i* iar ocuparea şi eliberarea au fost convenabil suprimate. Noua notaţie "::" se numeşte *subordonarea la distanţă*. Se observă diferenţa faţă de deosebit de familiara notaţie ":" prin aceea că în partea dreapta nu apare un proces complet ci numele unui masiv de procese ce se pot afla la distanţă.

X4 Două fişiere de ieşire
Un proces utilizator necesită utilizarea simultană a două imprimante prin transmiterea către ele a două fişiere de ieşire *f1* şi *f2*

$$(f1::lp/\!/(f2::lp/\!/\ldots f1.st\hat{a}nga!s1\rightarrow f2.st\hat{a}nga!s2\rightarrow \ldots))$$

Aici, procesul utilizator întreţese transmiterea liniilor către cele două fişiere diferite dar fiecare linie este tipărită pe imprimanta corespunzătoare. Desigur, blocajul va apare sigur la orice încercare de a declara *trei* imprimante (fizic tot două) simultan. De asemenea un rezultat asemănător apare în cazul când se declară două imprimante simultan în fiecare din cele două procese concurente, cum se vede bine din istoria lui Ann şi Mary (6.2 X5). □

X5 Fişier de lucru (manevră)
Un fişier de lucru este folosit la citire pentru transmiterea unei secvenţe de blocuri. Când transmisia este terminată, fişierul este rebobinat şi întreaga secvenţă de blocuri este citită din nou de la început. Când toate blocurile au fost citite, fişierul manevră va genera numai semnalul *gol*, nefiind posibile citirile sau scrierile ulterioare. Astfel, un fişier de manevră se comportă ca un fişier de lucru pe bandă magnetică care trebuie rebobinat înainte de a fi citit. Semnalul *gol* serveşte ca marker de end-of-file

$$MANEV = SCRIE$$
$$SCRIE_s = (st\hat{a}nga?x\rightarrow SCRIE_{s^\wedge\langle x\rangle} \mid rebobinare\rightarrow CITE\c{S}TE_s)$$
$$CITE\c{S}TE_{\langle x\rangle^\wedge s} = (dreapta!x\rightarrow CITE\c{S}TE_s)$$
$$CITE\c{S}TE_{\langle\rangle} = (gol\rightarrow CITE\c{S}TE_{\langle\rangle})$$

Acest proces poate fi utilizat convenabil ca un proces subordonat nepartajat

$$fispropriu:MANEV/\!/\ldots fispropriu.st\hat{a}nga!v\ \ldots fispropriu.rebobinare\ \ldots$$
$$\ldots (fispropriu.dreapta?x\rightarrow \ldots$$
$$\mid fispropriu.gol\rightarrow \ldots)\ \ldots)$$

Ne vom referi mai târziu la cele expuse mai sus ca la un model de proces partajat. □

X6 Fişiere de manevră în memoria secundară
Fişierul de manevră descris în X5 poate fi uşor implementat prin reţinerea secvenţei de blocuri în memoria principală (RAM) a computerului. Dar dacă blocurile sunt mari şi secvenţa lungă, aceasta ar fi o utilizare neeconomică a memoriei principale şi ar fi mai bine să memorăm blocurile într-o memorie secundară. Deoarece orice bloc într-un fişier de manevră este citit şi scris o singură dată, o memorie secundară (X2) cu citit destructiv va fi cea mai potrivită. Un fişier de lucru ordinar (ţinut în memoria principală) este utilizat pentru a reţine secvenţa de indici ale sectoarelor memoriei secundare unde se găsesc blocurile reale cu informaţie. Avem astfel certitudinea că sunt recitite corect blocurile, în secvenţa corespunzătoare

$$
\begin{aligned}
&SMANEV=(tabpag{:}MANEV/\!/\\
&\quad \mu X.(\text{stânga}?x\rightarrow(\underset{i<S}{\square}\,mems.i.\text{stânga}!x\rightarrow tabpag.\text{stânga}!i\rightarrow X)\\
&\qquad |\,rebobinare\rightarrow tabpag.rebobinare\rightarrow\\
&\qquad \mu Y.(tabpag.dreapta?i\rightarrow mems.i.dreapta?x\rightarrow dreapta!x\rightarrow Y\\
&\qquad\quad |\,tabpag.gol\rightarrow gol\rightarrow Y)))
\end{aligned}
$$

SMANEV foloseşte numele *mems* pentru a adresa o memorie secundară (X2) ca un proces subordonat. Putem descrie aceasta prin

$$MANEVS=(mems{:}SMEM/\!/SMANEV)$$

MANEVS poate fi utilizat ca un simplu proces subordonat nepartajat în exact acelaşi fel ca şi fişierul de lucru din X5

$$(fispropriu{:}MANEVS/\!/\ldots\ fispropriu.\text{stânga}!v\ \ldots)$$

Efectul este exact acelaşi ca şi la utilizarea lui *MANEV* cu deosebirea că lungimea maximă a fişierului de lucru este limitată la *S* blocuri. □

X7 Fişiere de manevră folosite serial
Să presupunem că vrem să partajăm fişierul de manevră din memoria secundară prin întreţeserea între mai mulţi utilizatori care îl vor solicita, utiliza şi elibera câte unul o dată, în maniera unei imprimante partajate (6.2 X3). Pentru acest scop trebuie să adaptăm *SMANEV* pentru a accepta semnalele *ocupă* şi *eliberează*. Dacă un utilizator eliberează fişierul înainte de a termina citirea există pericolul ca blocurile necitite din memoria secundară să nu mai fie accesate. Acest pericol poate fi evitat printr-o buclă care citeşte suplimentar aceste blocuri şi le eliberează (citire în gol)

$SCAN = \mu X.(tabpag.dreapta?i \rightarrow mems.i.dreapta?x \rightarrow X$
$\quad | tabpag.gol \rightarrow SKIP)$

Un fişier de manevră partajat întâi îşi captează utilizatorul şi apoi se comportă ca *SMANEV*. Semnalul *eliberează* provoacă o întrerupere (paragraful 5.4) procesului *SCAN*

$SMANPAR = ocupă \rightarrow (SMANEV \wedge (eliberează \rightarrow SCAN))$

Fişierul de manevră reutilizabil serial este modelat de bucla simplă

$*SMANPAR$

care foloseşte *SMEM* ca un proces subordonat

$mems{:}SMEM // *SMANPAR$ □

X8 Fişiere de manevră multiplexat
În cele două exemple anterioare, la un moment dat este utilizat numai un singur fişier de manevră. O memorie secundară este de obicei suficient de mare pentru a permite mai multor fişiere de manevră să coexiste simultan, fiecare ocupând o mulţime disjunctă de sectoare disponibile. Memoria secundară poate de aceea să fie partajată între un număr suficient de mare (nemărginit) de fişiere de manevră. Fiecărui fişier de manevră i se alocă un sector când are nevoie prin scrierea unui bloc de informaţii în el şi îl eliberează prin citirea informaţiei din acel sector. Memoria secundară este partajată prin tehnica etichetării multiple (paragraful 2.6.4) utilizând ca etichete aceiaşi indici (numere naturale) care sunt utilizaţi în construcţia unui masiv de procese partajante

$$SISFIS = \mathbf{N}{:}(mems{:}SMEM) // (\underset{i \geq 0}{|||}\, i{:}SMANPAR)$$

unde $\mathbf{N} = \{i \mid i \geq 0\}$.

Acest sistem de fişiere se intenţionează a se folosi ca un proces subordonat partajat, între orice număr de utilizatori întreţesuţi

$sisfis{:}SISFIS // \ldots (UTIL1 ||| UTIL2 ||| \ldots)$

Fiecare utilizator poate obţine un fişier manevră, utiliza şi elibera prin subordonare la distanţă

fispropriu::*sisfis*//(...*fispropriu.stânga*!v ... *fispropriu.rebobinare* ...
fispropriu.dreapta?x ...)

intenţionându-se (în afară de limitarea de resurse) obţinerea aceluiaşi efect ca şi prin subordonarea simplă a unui fişier manevră propriu (X5)

(*fispropriu*:*MANEV*//... *fispropriu.stânga*!v ... *fispropriu.rebobinare* ...
fispropriu.dreapta?x ...) □

Structura sistemului de fişiere (X8) şi modul său de utilizare este un model de soluţie generală a problemei partajării unui număr limitat de resurse reale (sectoare din memoria secundară) între un număr necunoscut de utilizatori. Utilizatorii nu comunică direct cu resursele ci există o *resursă virtuală* intermediară (*SMANPAR*) pe care ei (utilizatorii) o vor declara şi utiliza ca şi cum ar fi un proces subordonat propriu. Funcţia resursei virtuale este dublă :

(1) asigură o interfaţă clară şi simplă cu utilizatorul. În acest exemplu, *SMANPAR* reuneşte într-un singur fişier de manevră continuu o mulţime de sectoare dispersate din memoria secundară.
(2) garantează un acces într-adevăr disciplinat la resursele reale. De exemplu, *SMANPAR* asigură că fiecare utilizator citeşte numai din sectoarele alocate lui şi nu poate uita de eliberarea sectoarelor când se termină lucrul cu fişierul său de manevră.

Punctul (1) ne asigură că disciplina punctului (2) este mai puţin dureroasă.

Paradigma resurselor reale şi virtuale este foarte importantă în proiectarea unui sistem cu partajarea resurselor. Definiţia matematică a paradigmei este foarte complicată deoarece utilizează o mulţime nemărginită de numere naturale pentru implementarea creării dinamice necesară noilor procese virtuale şi noilor canale prin care să se comunice cu ele. Într-o implementare practică pe calculator, acestea ar putea fi reprezentate prin blocuri de control, pointeri la înregistrările completate, etc. Pentru a utiliza paradigma efectiv, este cu siguranţă mai bine să uităm metoda de implementare. Dar pentru acei care vor să o înţeleagă mai complet înainte de a o uita, următoarea explicaţie pentru X8 poate fi de ajutor.

Într-un proces utilizator un fişier manevră este creat printr-o subordonare la distanţă

fispropriu::*sisfis*//(... *fispropriu.stânga*!v ... *fispropriu.rebobinare* ...
fispropriu.dreapta?x ...)

Din definiţia subordonării la distanţă aceasta este echivalentă cu

($\square_{i\geq 0}$ *sisfis.i.ocupă*→
sisfis.i.stânga!v ... *sisfis.i.rebobinare* .. *sisfis.i.dreapta*?x ....
sisfis.i.eliberează→*SKIP*)

Astfel toate comunicaţiile între *sisfis* şi utilizatorii săi încep cu *sisfis.i...*, unde *i* este indexul instanţei particulare a lui *SMANPAR* care a fost obţinută de un utilizator oarecare într-o ocazie oarecare. Mai mult, fiecare ocazie a utilizării sale este încadrată de o pereche corespunzătoare de semnale

(*sisfis.i.ocupă*→ ... *sisfis.i.eliberează*)

Din punctul de vedere al procesului subordonat, fiecare fişier de manevră virtual începe cu obţinerea utilizatorului său şi continuă potrivit schemei specificate în X6 şi X7

(*ocupă*→*stânga*?x ... *rebobinare* ... *dreapta*!v ... *eliberează*)

Toate celelalte comunicaţii ale fişierului de manevră virtual sunt cu procesul subordonat *SMEM* şi sunt mascate utilizatorului. Fiecare instanţă a fişierului de manevră virtual este indexat cu un index diferit *i* şi apoi numit cu numele *sisfis*. Astfel comportarea vizibilă externă a fiecărei instanţe este

(*sisfis.i.ocupă*→
sisfis.i.stânga?x ... *sisfis.i.rebobinare* ... *sisfis.i.dreapta*!v ...
sisfis.i.eliberează)

Aceasta se potriveşte exact schemei de comunicare a utilizatorului descrisă în paragraful precedent. Perechile corespunzătoare de semnale *ocupă* şi *eliberează* ne asigură că nici un utilizator nu poate interfera cu un fişier de manevră care a fost obţinut de un alt utilizator.

Trecem acum la comunicaţiile din *SISFIS* între masivul de fişiere de manevră virtuale şi memoria secundară. Acestea sunt mascate utilizatorului şi nici măcar nu au numele *sisfis* ataşat lor. Evenimentele relevante sunt

*i.mems.j.stânga.*v	înseamnă comunicarea unui bloc v de la elementul *i* din masivul de fişiere de manevră către sectorul *j* din memoria secundară.
*i.mems.j.dreapta.*v	înseamnă comunicarea în direcţia inversă.

Fiecare sector al memoriei secundare se comportă ca procesul *COPIE*. După indexarea cu un număr de sector *j* şi numirea cu *mems*, sectorul *j* se comportă ca

$\mu X.(mems.j.stânga?x \rightarrow mems.j.dreapta!x \rightarrow X)$

După etichetarea multiplă cu numere naturale avem comportarea

$\mu X.(\underset{i \geq 0}{\square} i.mems.j.stânga?x \rightarrow (\underset{i \geq 0}{\square} k.mems.j.dreapta!x \rightarrow X))$

Acest proces este acum gata de a comunica în orice ocazie cu orice element al masivului de fişiere de manevră virtuale. Fiecare fişier individual respectă disciplina citirii numai din acele sectoare în care s-a scris recent.

În descrierea de mai sus, rolul numerelor naturale *i* şi *j* este mai mult de a permite ca orice fişier de manevră să comunice cu orice sector de pe disc şi totodată să comunice în siguranţă cu utilizatorul care l-a obţinut. Indicii se pot asemăna de aceea cu descrierea matematică a unui comutator multiplu, care este folosit în telefonie pentru a permite conectarea între diverşi abonaţi. Un desen simplificat este în fig. 6.2.

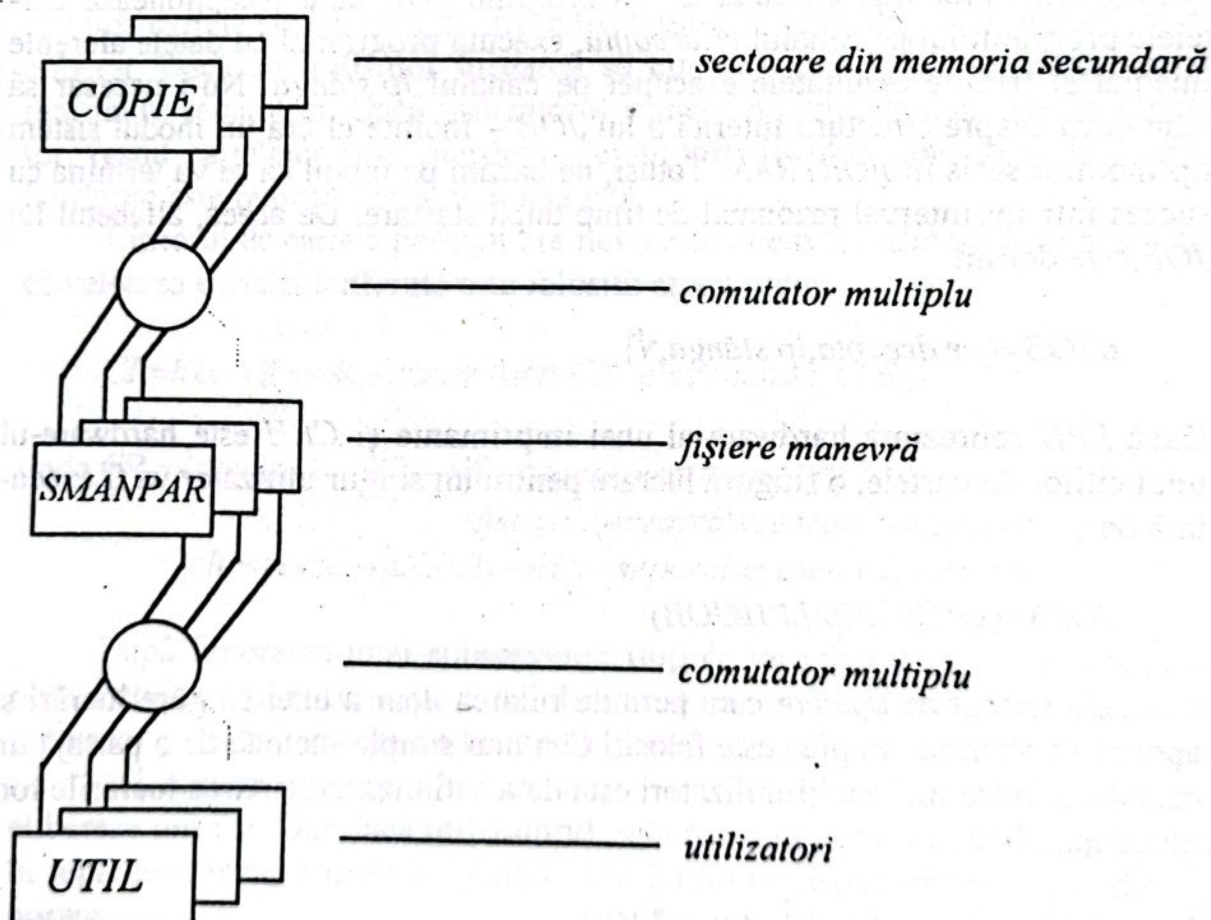


Figura 6.2

Dacă numărul sectoarelor din mediul de memorare secundar este infinit, *SISFIS* se comportă exact ca un masiv de fişiere de manevră simple

$$\|_{i\geq 0} i{:}(ocupă \rightarrow (MANEV^\wedge(eliberează \rightarrow STOP)))$$

Cu o memorie secundară de dimensiune finită există pericolul de blocaj în momentul când memoria se umple şi toţi utilizatorii continuă să scrie mai departe în fişierele lor de manevră. În realitate, acest risc este de fapt redus semnificativ prin punerea în aşteptare a obţinerii de noi fişiere când memoria este aproape plină.

6.5 Sisteme de operare

Utilizatorii unui singur calculator mare îşi prezintă programele pentru execuţie sub formă de pachete de cartele. Datele pentru fiecare program urmează imediat. Sarcina unui sistem de operare de tip batch este de a partaja resursele calculatorului eficient între aceste *job-uri*. De aceea precizăm că programul fiecărui utilizator este executat de un program *JOB* care recepţionează cartelele programului pe canalul *cr.dreapta*, execută programul cu datele aferente imediat şi trimite rezultatele execuţiei pe canalul *lp.stânga*. Nu-i necesar să ştim ceva despre structura internă a lui *JOB* – înainte el era un modul sistem tip monitor scris în *FORTRAN*. Totuşi, ne bazăm pe faptul că se va termina cu succes într-un interval rezonabil de timp după startare. De aceea, alfabetul lui *JOB* este definit

$$\alpha JOB=\{cr.dreapta, lp.stânga, \surd\}$$

Dacă *LPH* reprezintă hardware-ul unei imprimante şi *CRH* este hardware-ul unui cititor de cartele, o singură lucrare pentru un singur utilizator va fi executată de

$$JOB1=(cr{:}CRH /\!/ lp{:}LPH /\!/ JOB)$$

Un sistem de operare care permite rularea doar a unei singure lucrări şi apoi să se termine nu prea este folosit. Cea mai simplă metodă de a partaja un calculator între mai mulţi utilizatori este de a serializa executarea lucrările lor, una după alta

$$BATCH0=(cr{:}CRH /\!/ lp{:}LPH /\!/ {}^{*}JOB)$$

Dar această proiectare ignoră câteva detalii administrative importante, ca de exemplu separarea transmiterii către imprimante a fişierelor pentru fiecare lucrare şi separarea citirii pachetelor de cartele conţinând o lucrare de cea anterioară, astfel că o lucrare să nu poată îngloba cartelele succesoarei sale. Pentru

a rezolva aceste probleme folosim procesul *LP* definit în 6.2 X4 şi un proces *CR* definit mai jos (X1)

$JOBS = {}^{*}((cr.ocupă \rightarrow lp.ocupă \rightarrow JOB);$
$\quad (cr.eliberează \rightarrow lp.eliberează \rightarrow SKIP))$
$BATCH1 = (cr{:}CR // lp{:}LP // JOBS)$

*BATCH*1 este o descriere abstractă a celui mai simplu şi viabil sistem de operare, partajând un calculator între mai mulţi utilizatori ale căror lucrări sunt executate rând pe rând succesiv. Sistemul de operare realizează tranziţia între lucrări succesive şi protejează fiecare lucrare de posibile interferenţe cu predecesoarele sale.

Exemple

X1 Un cititor de cartele partajat
O cartelă specială separator este introdusă la începutul fiecărei lucrări. Cititorul de cartele este afectat pentru citirea tuturor cartelelor unei lucrări şi apoi eliberat. Dacă un utilizator încearcă să citească dincolo de separator, există mai mulţi separatori. Dacă utilizatorul eşuează înainte de a ajunge la separator, restul cartelelor sunt pierdute. Separatorii în plus sunt ignoraţi. Citirea hardware este realizată prin acţiunea $h?x$.

Cititorul de cartele partajat are nevoie să citească o cartelă în avans astfel că valoarea cartelei bufferate este folosită ca un index.

$$CR = h?x \rightarrow \textbf{if } x = separator \textbf{ then } CR \textbf{ else } (ocupă \rightarrow CR_x)$$

unde $CR_x = (dreapta!x \rightarrow h?y \rightarrow \textbf{if } y \neq separator \textbf{ then } CR_y$
$\quad \textbf{else } \mu X.(dreapta!separator \rightarrow X \mid eliberează \rightarrow CR)$
$\quad \mid eliberează \rightarrow \mu X.(h?y \rightarrow \textbf{if } y = separator \textbf{ then } CR \textbf{ else } X))$

După ignorarea unei subsecvenţe iniţiale de separatori, acest proces îşi obţine utilizatorul şi copie pe canalul din dreapta secvenţa cartelelor diferite de separator pe care le citeşte de la canalul hardware. Când se detectează un separator, valoarea sa este reprodusă de atâtea ori cât este necesar ca utilizatorul să elibereze resursa. Dar dacă utilizatorul eliberează cititorul înainte să se ajungă la separator, restul cartelelor până la următorul separator trebuie citite dar ignorate. □

Sistemul de operare *BATCH*1 este logic complet. Totuşi, cu cât hardware-ul calculatorului funcţionează mai repede se depăşeşte capacitatea cititoarelor şi imprimantelor de a primi la intrare şi a transmite la ieşire mai multe lucrări. Pentru a stabili o legătură între intrare, ieşire şi viteza de proce-

sare, este necesar şi raţional să folosim două sau mai multe cititoare şi imprimante. Deoarece o singură lucrare este procesată odată, cititoarele suplimentare ar putea citi în avans fişierul de cartele pentru următoarea lucrare sau lucrări şi imprimantele suplimentare ar putea imprima fişierul de ieşire a lucrării anterioare. De aceea, fiecare fişier de intrare trebuie ţinut într-un fişier temporar de la citirea efectivă cu cititorul şi până la prelucrarea în *JOB*. Similar, fiecare fişier de ieşire trebuie reţinut de la producerea lui în *JOB* şi până la imprimarea efectivă pe o imprimantă. Această tehnică se numeşte *organizarea în coadă* (*spooling*).

Stuctura generală a unui sistem de operare cu cozi este

*SISOP*1=*sisin*:*COADĂIN*//*sisout*:*COADĂOUT*//*BATCH*

Aici *BATCH* este ca şi *BATCH*1 cu deosebirea că foloseşte subordonarea la distanţă pentru a obţine oricare din fişierele de intrare în aşteptare şi de asemenea de a obţine un fişier de ieşire care este destinat imprimării următoare

BATCH=*((*cr*::*sisin*//*lp*::*sisout*//*JOB*)

Procesele care gestionează cozile de intrare şi ieşire sunt definite în următoarele două exemple.

X2 Coadă de ieşire

O singură imprimantă virtuală foloseşte un fişier de manevră temporar (6.4 X5) pentru a reţine blocurile ce trebuie imprimate, generate de procesul său utilizator. Când procesul utilizator semnalează eliberarea imprimantei virtuale, atunci o imprimantă adevărată este obţinută (6.4 X3) pentru a imprima conţinutul fişierului temporar.

VLP=(*temp*:*MANEV*//
  μ*X*.*stânga*?*x*→*temp*.*stânga*!*x*→*X*
    |*eliberează*→*temp*.*rebobinare*→
    (*real*::*lp*//
    μ*Y*.(*temp*.*dreapta*?*y*→*real*.*stânga*!*y*→*Y*
      |*temp*.*gol*→*SKIP*)))

Tabloul nemărginit de imprimante virtuale necesare este definit

$VLPS = \|_{i \geq 0}\, i{:}(ocupă \rightarrow VLP)$

Deoarece dorim ca imprimantele reale (6.4 X3) să fie folosite numai în modul *coadă* le putem declara locale sistemului ce folosește etichetare multiplă, pentru a le partaja între toate elementele masivului *VLPS* ca în 6.4 X8

COADĂOUT=(**N**:(*lp*:*LP*2)//*VLPS*) □

X3 Coadă de intrare

Coada de intrare este foarte asemănătoare celei de ieșire cu deosebirea că un cititor de cartele real este întâi afectat și apoi eliberat la sfârșitul intrării pentru o singură lucrare. Un proces utilizator este apoi afectat să execute lucrarea și conținutul cartelelor este transmis lui.

VCR=*temp*:*MANEV*//
(*real*::*cr*//
(μ*X*.*real*.*dreapta*?*x*→**if** *x*=*separator* **then** *SKIP*
else *temp*.*stânga*!*x*→*X*));
(*temp*.*rebobinare*→*ocupă*→
(μ*Y*.(*temp*.*dreapta*?*x*→*dreapta*!*x*→*Y*
|*temp*.*gol*→*dreapta*!*separator*→*Y*))^(*eliberează*→*SKIP*))

COADĂIN=(**N**:*cr*:(0:*CR*||1:*CR*))//(||*i*:*VCR*) □
$\scriptstyle i\geq 0$

Cozile de intrare și ieșire alimentează acum un număr nelimitat de cititoare și imprimante virtuale pentru folosirea de către procesul *JOB*. Ca un prim rezultat, este posibil pentru două sau mai multe procese să evolueze concurent, partajând aceste resurse virtuale. Deoarece nu se cere vreo comunicație între lucrări, întrețeserea simplă este cea mai potrivită metodă de partajare. Această tehnică este cunoscută ca *multiprogramare*, sau dacă sunt folosite mai mult decât un procesor, este cunoscută ca *multiprocesare*. Totuși efectele logice ale multiprogramării și multiprocesării sunt aceleași. Într-adevăr, sistemul de operare definit mai jos are aceleași specificații logice ca și *SISOP*1 definit mai sus.

SISOP=*sisin*:*COADĂIN*//*sisout*:*COADĂOUT*//*BATCH*4
unde *BATCH*4=(|||*BATCH*)
$\scriptstyle i<4$

Matematic vorbind, trecerea la multiprogramare s-a făcut remarcabil de simplu : din punct de vedere istoric însă a cauzat probleme mari.

În proiectarea procesului *VLP* din *COADĂOUT* (X2), procesul subordonat *MANEV* a fost folosit pentru a reține liniile pentru imprimare produse de fiecare job până când sunt imprimate efectiv pe o imprimantă reală. În ge-

neral, fişierele de ieşire sunt prea mari pentru a fi reţinute în memoria principală a unui computer, de aceea ele se reţin în memoria secundară, cum s-a arătat în 6.4 X8. Dacă toate fişierele temporare partajează aceeaşi memorie secundară, trebuie să înlocuim procesul subordonat

temp:MANEV//...

din *VLP* printr-o declaraţie a unui proces subordonat la distanţă.

temp::sisfis//...

şi apoi să declarăm sistemul (6.4 X8) ca un proces subordonat al cozii de ieşire

(*sisfis:SISFIS//COADĂOUT*)

Dacă volumul de date al unei cartele este semnificativ, trebuie făcută o schimbare similară la *COADĂIN*. Dacă o memorie secundară separată este disponibilă pentru acest scop, schimbarea este uşoară. Dacă nu, va trebui să partajăm aceeaşi memorie secundară între fişierele atât ale cozilor de intrare cât şi ale celor de ieşire. Aceasta înseamnă că *SISFIS* trebuie declarat ca un proces subordonat, partajat prin etichetare multiplă între cozi, aceasta implicând o schimbare în structura sistemului. Vom face această reproiectare într-o manieră *top-down*, încercând să folosim cât mai multe din modulele definite.

Sistemul de operare este compus dintr-un sistem cu multiprogramare *BATCH*4 şi un sistem de intrare-ieşire servind ca proces subordonat

*OP=SISTEMIO//BATCH*4

Sistemul de intrare-ieşire partajează un sistem de fişiere între coada de intrare şi cea de ieşire

SISTEMIO=PART:(sisfis:SISFIS)
//(lp:COADĂOUT' ||cr:COADĂIN')

şi $PART=\{lp.i \mid i\geq 0\}\cup\{cr.i \mid i\geq 0\}$

iar *COADĂOUT'* şi *COADĂIN'* sunt aceleaşi ca în X2 şi X3, cu deosebirea că

temp:MANEV

este înlocuit de echivalentul lui subordonat la distanţă

temp::sisfis

În proiectarea celor patru sisteme de operare descrise în acest capitol (*BATCH*1, *SISOP*1, *SISOP* şi *OP*) am pus accentul mai presus de orice pe modularitate. Aceasta înseamnă că am putut refolosi mari părţi din sistemele anterioare în cele ulterioare. Chiar mai important, orice decizie de detaliu este izolată în unul sau mai multe module din sistem. Ulterior, dacă trebuie schimbat un detaliu, este foarte uşor să identificăm care modul trebuie alterat şi schimbările pot fi delimitate la acel modul. Printre modificările uşoare sunt

numărul imprimantelor
numărul cititoarelor de cartele
numărul de loturi concurente

Dar nu toate modificările vor fi la fel de uşoare : o schimbare a valorii cartelei separator va afecta trei module *CR* (X1), *COADĂIN* (X3) şi *JOB*.

Sunt de asemenea un număr de îmbunătăţiri valorice ale sistemului care ar necesita schimbări semnificative în structura sa.

(1) Joburile utilizator să aibă de asemenea acces la sistemul de fişiere şi la dispozitivele virtuale multiple de intrare şi ieşire.
(2) Fişierele utilizatorilor să poată fi permanent reţinute între joburile ce le utilizează.
(3) Ar fi necesară o metodă de puncte de control pentru reluare rapidă în caz de cădere.
(4) Dacă există un număr de joburi de intrare în aşteptare (o listă) dar neexecutate, este nevoie de o metodă de a controla ordinea în care joburile în aşteptare sunt lansate. Această chestiune este tratată mai complet în paragraful următor.

Una din problemele întâlnite când s-au făcut îmbunătăţiri este imposibilitatea partajării resurselor între un proces subordonat şi procesul principal corespunzător, în acele cazuri în care tehnica de etichetare multiplă nu este adecvată. Pare că o nouă definiţie a subordonării este necesară, în care alfabetul procesului subordonat să nu fie o submulţime a alfabetului procesului principal. Dar aceasta este o problemă pentru o cercetare ulterioară.

6.6 Planificare

Când un număr de resurse limitate este partajat între un număr mare de potenţiali utilizatori va exista totdeauna posibilitatea ca anumiţi utilizatori doritori de resurse să fie nevoiţi să aştepte obţinerea uneia până ce un alt proces o eliberează. Dacă în momentul eliberării sunt două sau mai multe procese în

aşteptarea obţinerii resursei, alegerea unuia din ele este nedeterministă ca în toate exemplele date. În sine, aceasta situaţie nu ridică probleme. Dar să presupunem că tocmai în momentul când resursa este eliberată procesul doritor se alătură mulţimii proceselor în aşteptare. Deoarece alegerea între procesele în aşteptare este nedeterministă s-ar putea ca ultimul proces să fie cel ales. Dacă resursa necesită o tratare prelungită aceasta se poate întâmpla mereu. Ca rezultat, unele din procese pot fi întârziate pentru o perioadă neprevizibilă şi neacceptabilă de timp. Aceasta este aşa-numita problemă a *preluării infinite* (*overtaking*) (paragraful 2.5.5).

O soluţie ar fi ca toate resursele să necesite o tratare cât mai scurtă. Aceasta se poate asigura prin multiplicarea resurselor, prin raţionalizarea utilizării lor sau prin impunerea unui preţ mare pentru utilizarea serviciilor asigurate. De fapt acestea sunt singurele soluţii în cazul unei resurse ce necesită o tratare complexă. Din nefericire, chiar o resursă care are o medie de ocupare (încărcare) uşoară ar putea fi intens folosită pentru perioade lungi (momente de vârf sau aglomeraţie). Problema poate fi rezolvată prin încercarea de a netezi cererea dar aceasta nu este totdeauna posibil sau nu poate fi făcut cu succes. În timpul momentelor de vârf este inevitabil ca în medie procesele utilizator să fie supuse întârzierilor. Este important de asigurat ca aceste momente de întârziere să fie în mod rezonabil previzibile şi consistente – se prefera mai mult să se ştie că vei fi servit într-o oră decât să mă întreb dacă va trebui să aşteptăm un minut sau o zi.

Activitatea de decizie a alocării unei resurse între mai mulţi utilizatori în aşteptare este cunoscută ca *planificare (scheduling)*. Pentru a planifica cu succes este necesar să ştiu care procese sunt în aşteptarea resursei. Din această cauză, obţinerea unei resurse nu mai poate fi privită ca un eveniment atomic. Ea trebuie despărţită în două evenimente

vă_rog care presupune cererea de alocare
mulţumesc care înseamnă alocarea de fapt a resursei.

Pentru orice proces, perioada dintre *vă_rog* şi *mulţumesc* este perioada de aşteptare a resursei. Pentru a identifica procesul care cere, vom indexa fiecare apariţie a lui *vă_rog*, *mulţumesc* şi *eliberează* cu un număr natural diferit. Procesul care cere o resursă o va obţine de fiecare dată prin secvenţa corespunzătoare numărului său, aceeaşi construcţie ca şi subordonarea la distanţă (6.4 X3)

$$\square_{i \geq 0}(cer.i.vă_rog;\ cer.i.mulţumesc;\ \dots\ ;\ cer.i.eliberează \rightarrow SKIP)$$

O metodă simplă şi efectivă de planificare a resurselor este de a aloca ce este disponibil procesului care a aşteptat cel mai mult. Această politică este cunoscută ca *FCFS* (primul venit primul servit) sau *FIFO* (primul intrat

primul ieşit). Este o disciplină de coadă care se observă la pasagerii dintr-o coadă de autobuz.

Într-o brutărie particulară unde pâinea se face una câte una şi unde consumatorii sunt în imposibilitate sau nu vor să formeze o coadă, există un mecanism alternativ pentru a realiza acelaşi efect (problema mai este cunoscută ca problema cofetăriei, patiseriei, etc. *n.t.*). S-a instalat o maşină care generează tichete cu o ordine serială ascendentă strictă în ce priveşte numerele. La intrarea în brutărie un cumpărător primeşte un tichet. Când un lucrător este gata, el cheamă cumpărătorul cu cel mai mic număr de tichet care n-a fost încă servit. Presupunem că până la C cumpărători pot fi serviţi simultan.

Exemplu

X1 Algoritmul brutăriei

Trebuie să ţinem următoarele socoteli

r - clienţi care au spus *vă_rog*
m - clienţi care au spus *mulțumesc*
e - clienţi care au eliberat resursele

În mod clar, totdeauna $e \leq m \leq r$. De asemenea, totdeauna r va fi numărul dat următorului client care intră în brutărie şi m este numărul următorului client de servit. Mai mult, $r-m$ este numărul clienţilor în aşteptare şi $C+e-m$ este numărul lucrătorilor în aşteptare. Toţi contorii sunt iniţial 0 şi pot reveni la 0 oricând sunt egali – de exemplu noaptea după ce a plecat ultimul client.

Una din sarcinile principale ale algoritmului este să asigure că nu există simultan o resursă liberă şi un client aşteptând. Ori de câte ori este o astfel de situaţie, următorul eveniment trebuie să fie *mulțumesc* de la un client obţinând resursa

$$
\begin{aligned}
&BRUT\breve{A}RIE = B_{0,0,0} \\
&B_{r,m,e} = \textbf{if } 0<e=m=r \textbf{ then } BRUT\breve{A}RIE \\
&\qquad \textbf{else if } C+e-m>0 \wedge r-m>0 \\
&\qquad\qquad \textbf{then } m.mul\c{t}umesc \rightarrow B_{r,m+1,e} \\
&\qquad\qquad \textbf{else}(r.v\breve{a}_rog \rightarrow B_{r+1,m,e} \\
&\qquad\qquad\quad | (\underset{i<m}{\square} i.elibereaz\breve{a} \rightarrow B_{r,m,e+1}))
\end{aligned}
$$

Algoritmul brutăriei se datorează lui Leslie Lamport. □

7 Discuție

7.1 Introducere

Principalul obiectiv al acestei lucrări din domeniul proceselor comunicante a fost găsirea celei mai simple posibile teorii matematice cu următoarele proprietăți :.

(1) Să poată descrie o gamă cât mai largă de aplicații pe calculator, de la automate de vândut, controlul proceselor și simularea cu evenimente discrete, până la sisteme de operare cu resurse partajabile.

(2) Să fie capabilă de o implementare eficientă pe o varietate de arhitecturi convenționale și de ultimă oră, de la calculatoare cu time-sharing, multiprocesoare, până la rețele de microprocesoare ce comunică între ele.

(3) Să asigure un sprijin clar programatorului în sarcinile sale de specificare, proiectare, implementare, verificare și validare a unor sisteme complexe.

Este evident că nu putem cere ca toate aceste obiective să fie realizate de o manieră optimă. Există însă totdeauna speranța că o metodă diferită radical de cele existente sau câteva schimbări importante în detalierea definițiilor ne-ar conduce la un succes mai mare corespunzător unuia sau mai multor din obiectivele enumerate mai sus. Acest capitol încearcă o discuție a unor alternative explorate de autor împreună cu alții și o explicație de ce această teorie și nu alta. De asemenea, dă ocazia autorului să recunoască influența cercetării originale a altor savanți în domeniu. În fine, se speră încurajarea cercetării ulterioare privind bazele acestei teorii precum și larga sa aplicabilitate practică.

7.2 Partajarea memoriei

Primele propuneri din 1960 privind programarea concurentă a activităților într-un calculator provin natural din dezvoltările contemporane ale arhitecturii

calculatoarelor și sistemelor de operare. În acele timpuri puterea de procesare era insuficientă și scumpă și era considerat pierdere de timp ca un procesor să aștepte comunicația cu un echipament periferic mai lent sau chiar cu utilizatorul. De aceea, ulterior s-au dezvoltat controllere speciale (procesoare de canal) pentru operații de I/O independente permițând astfel procesorului central să se angajeze în alte acțiuni. Pentru a ține ocupat procesorul central, un sistem de operare cu partajarea timpului trebuia să asigure existența simultană a unor programe în memoria principală a calculatorului. Astfel, în orice moment orice program putea accesa controllere de I/O în timp ce un alt program putea folosi procesorul principal. La terminarea operației de I/O o întrerupere permitea sistemului de operare reconsiderarea programului ce a generat accesul astfel ca acesta să se bucure în continuare de atenția procesorului central.

Schema descrisă mai sus sugerează că procesorul central și toate procesoarele de canal ar trebui conectate la memoria principală a calculatorului determinând ca accesele la memorie să fie întrețesute. Cu toate acestea, fiecare program executat era de obicei o lucrare completă afectată unui utilizator anume și complet independentă de alte lucrări.

Din acest motiv exista o mare grijă în proiectarea hardware-ului și software-ului pentru împărțirea memoriei în segmente disjuncte, unul pentru fiecare program, cu asigurarea că nici un program nu putea interfera cu altul. Când a devenit posibil de atașat mai multe procesoare independente în același calculator efectul a fost creșterea răspunsului lucrărilor (joburilor). Dacă sistemele de operare originale erau bine structurate acest lucru se putea realiza cu puține schimbări în sistemul de operare și chiar și mai puține în programele pentru lucrările ce se executau.

Dezavantajele partajării unui calculator între mai multe lucrări distincte erau

(1) Dimensiunea memoriei necesare creștea liniar cu numărul de lucrări executate simultan
(2) Timpul necesar fiecărui utilizator pentru obținerea rezultatelor la lucrarea sa creștea de asemenea cu excepția lucrărilor de cea mai mare prioritate.

De aceea pare rezonabil de a permite unei singure lucrări să profite de paralelismul asigurat de hardware-ul calculatorului prin inițierea mai multor procese concurente în zona de memorie alocată unui singur program.

7.2.1 Multiplicarea firelor de execuție

Prima propunere de acest fel s-a bazat pe un salt (comandă **go to**). Dacă *L* este o etichetă din program, comanda

fork *L*

transferă controlul etichetei *L* şi *de asemenea* permite executarea următoarei comenzi din secvenţă. Din acel moment efectul este ca şi cum două procesoare execută acelaşi program în acelaşi timp. Fiecare îşi menţine însă firul său de execuţie. Deoarece fiecare fir de control poate produce din nou un *fork* această tehnică de programare este cunoscută ca *multiplicarea firelor de execuţie*.

Odată dovedită o metodă ca un proces să poată fi descompus în două este necesară o metodă ca două procese să poată fi unificate. O propunere simplă este asigurarea unei comenzi

join

care poate fi realizată numai când *două* procese o execută simultan. Primul proces care ajunge la comandă trebuie mai întâi să aştepte până ce un altul ajunge de asemenea. După aceea cele două devin unul singur.

Datorită generalităţii sale, multiplicarea firelor de execuţie este o tehnică incredibil de complexă şi predispusă la erori, nefiind recomandată decât în programe mici. Ca o scuză ea a fost inventată înainte de a apare programarea structurată, când chiar FORTRAN-ul era considerat un limbaj de programare de nivel înalt !

O variantă a comenzii *fork* este încă utilizată în sistemul de operare UNIX™. *Fork*-ul nu foloseşte nici o etichetă. Efectul ei este de a face o copie a întregii memorii alocate programului, copia devenind un nou proces. Atât procesul original cât şi copia îşi continuă execuţia cu următoarea comandă de după *fork*. Este asigurată de asemenea o facilitate pentru fiecare program de a descoperi dacă este *părinte* sau *fiu*. Alocarea de zone disjuncte de memorie proceselor elimină principalele dificultăţi şi pericole ale tehnicii multiplicării firelor de execuţie dar poate fi ineficientă atât în timp cât şi în spaţiu. Aceasta înseamnă că este permisă concurenţa numai la nivelul global al unui job iar utilizarea sa la o scară mai mică este descurajată.

7.2.2 Cobegin ... coend

O soluţie la problema multiplicării firelor de execuţie a propus-o E. W. Dijkstra prin asigurarea că după un *fork* cele două procesoare execută blocuri de program complet diferite fără posibilitatea saltului între ele. Dacă *P* şi *Q* sunt astfel de blocuri, comanda compusă

cobegin *P*; *Q* **coend**

determină ca *P* şi *Q* să pornească simultan şi să se execute concurent până ce ambele s-au terminat. După aceasta rămâne numai un singur procesor să exe-

cute mai departe alte comenzi. Această comandă structurată poate fi implementată de comenzile nestructurate *fork* şi *join*, folosind etichetele *L* şi *J*

fork *L*; *P*; **go to** *J*; *L*: *Q*; *J*: **join**

Generalizarea la mai mult de două procese componente este imediată şi evidentă

cobegin *P*; *Q*; ... ;*R* **coend**

Un mare avantaj al acestei notaţii structurate este uşurinţa de a înţelege ce se întâmplă de fapt, în special dacă variabilele folosite în fiecare din blocuri sunt distincte de variabilele folosite în altele (o restricţie care poate fi verificată sau impusă de un compilator pentru un limbaj de nivel înalt). În acest caz procesele se spune că sunt *disjuncte* şi (în absenţa comunicaţiei) execuţia concurentă a lui *P* şi *Q* are exact acelaşi efect ca şi execuţia lor secvenţială în altă ordine

begin *P*; *Q* **end** = **begin** *Q*; *P* **end** = **cobegin** *P*; *Q* **coend**

Mai mult, metodele de demonstrare pentru stabilirea corectitudinii compunerii paralele pot fi chiar mai simple decât cazul secvenţial. Din această cauză propunerea lui Dijkstra este baza construcţiei paralele din această carte. Principala schimbare este notaţia. Pentru a evita confuzia cu compunerea paralelă eu am introdus între procese operatorul || permiţându-se astfel utilizarea parantezelor pentru includerea comenzilor în loc de scrierea mai stânjenitoare **cobegin** ... **coend**.

7.2.3 Regiuni critice condiţionale

Restricţia ca procesele concurente să nu partajeze variabile are drept consecinţă faptul că ele nu pot comunica sau interacţiona între ele în vreun fel, o restricţie care reduce substanţial valoarea potenţială a concurenţei.

După citirea acestei cărţi introducerea canalelor de intrare şi ieşire (simulate) poate părea soluţia evidentă. La începuturi o tehnică evidentă (sugerată de hardware-ul calculatoarelor) era de a comunica prin partajarea memoriei principale între procesele concurente. Dijkstra a arătat cum se putea realiza aceasta relativ fără griji cu ajutorul regiunilor critice (paragraful 6.3.) protejate de semafoare de excludere mutuală. Eu am propus mai târziu ca această metodă să fie formalizată prin notaţiile unui limbaj de nivel înalt. Un grup de variabile care trebuie actualizate în regiunile critice din mai multe procese partajante s-au declarat ca resurse partajate, de exemplu

shared *n* : *integer*;
shared *poziție* : **record** *x,y* : *real* **end**

Fiecare regiune critică care actualizează această variabilă este precedată de o clauză **with**, indicând numele variabilei

with *n* **do** *n* : = *n* + 1;
with *poziție* **do begin** *x* : = *x* + *deltax*; *y* : = *y* + *deltay* **end**

Avantajul acestei notații este acela că un compilator introduce automat semafoarele necesare și delimitează fiecare regiune critică prin operațiile necesare *P* și *V*. Mai mult, poate verifica la compilare dacă vreo variabilă partajată poate fi accesată sau actualizată într-o regiune critică protejată de semafoarele corespunzătoare.

Cooperarea între procesele concurente care partajează memorie necesită de multe ori alte forme de sincronizare. De exemplu, un proces actualizează o variabilă în ideea că alte procese ar putea citi noua variabilă. Celelalte procese nu trebuie să citească variabila până ce actualizarea n-a avut loc. Similar, primul proces nu trebuie să actualizeze variabila până ce celelalte procese n-au citit versiunea anterioară.

Pentru a rezolva această problemă o facilitate convenabilă este oferită prin regiunea critică condițională. Aceasta ia forma

with *var_part* **when** *condiție* **do** *regiune critică*

La intrarea în regiunea critică este testată valoarea condiției. Dacă este adevărată, regiunea critică este executată normal. Dacă condiția este falsă această intrare în regiunea critică este amânată astfel că altor procese li se permite intrarea în regiunile lor critice și actualizarea variabilei partajate. La terminarea actualizărilor condiția este retestată. Dacă a devenit adevărată, procesul amânat poate pătrunde în regiunea sa critică, altfel procesul este suspendat din nou, iar dacă trebuie făcută o alegere între mai mult de un proces amânat, alegerea este arbitrară.

Pentru a rezolva problema actualizării și citirea mesajului de mai multe procese se declară ca făcând parte din resursă o variabilă întreagă, contor pentru un număr de procese care trebuie să citească mesajul înainte ca el să fie actualizat din nou

shared *mesaj* : **record** *contor* : *integer*; *conținut* :... **end**;
mesaj.contor : = 0;

Procesul care actualizează conține o regiune critică de forma

```
with mesaj when contor = 0 do
   begin conținut : = ... ;
      ...;
      contor : = număr de cititori
   end
```

Fiecare proces cititor conţine o regiune critică de forma

```
with mesaj when contor > 0 do
   begin copia personală : = conținut; contor : = contor–1 end
```

Regiunile critice condiţionale pot fi implementate cu ajutorul semafoarelor. Comparativ cu utilizarea directă a semafoarelor de sincronizare de către programator, plusul adus de regiunile critice condiţionale este destul de mare deoarece condiţiile tuturor proceselor în aşteptare la intrarea în regiunea critică trebuie retestate la fiecare ieşire din regiune. Din fericire, condiţiile nu trebuie testate mai frecvent decât după o ieşire deoarece restricţiile impuse accesului la variabilele partajate ne asigură că o condiţie testată de un proces în aşteptare, poate fi schimbată ca valoare numai dacă variabila partajată însăşi îşi schimbă valoarea. Toate celelalte variabile din condiţie trebuie să fie particulare procesului în aşteptare care evident nu le poate schimba când este în aşteptare.

7.2.4 Monitoare

Dezvoltarea monitoarelor s-a inspirat din ideea de *clasă* (*class*) din SIMULA 67, care la rândul ei era o generalizare a conceptului de procedură din ALGOL 60. Principala lor caracteristică este aceea că toate operaţiile importante cu date (incluzând iniţializarea) ar putea fi reunite împreună într-o declaraţie de structură şi tip de dată. Aceste operaţii ar putea fi invocate prin apeluri de procedură ori de câte ori sunt necesare proceselor care partajează datele. Caracteristica importantă a unui monitor este că numai una din procedurile corpului său poate fi activă la un moment dat. Atunci când două procese apelează simultan o procedură (aceeaşi procedură sau diferite), unul din apeluri este întârziat până când celălalt este terminat. Astfel corpurile procedurilor joacă rolul regiunilor critice condiţionale protejate de acelaşi semafor. De exemplu, un semafor foarte simplu se comportă ca o variabilă contor. În notaţia PASCAL PLUS avem

```
monitor contor;
var n : integer;
procedure *incr; begin n : = n + 1 end;
procedure *decr; when n > 0 do begin n := n–1 end;
```

```
5 function *zero : Boolean; begin zero := (n = 0) end;
6 begin n := 0;
7    . . . ;
8    if n ≠ 0 then print (n)
9 end
```

Linia 1 declară monitorul şi îi dă numele *contor*.

2 declară variabila partajată *n* locală monitorului. Ea este inaccesibilă în afara monitorului.

3
4
5 } declară trei proceduri cu corpurile lor. Asteriscul ne asigură că ele pot fi apelate numai din programul care utilizează monitorul.

6 Monitorul porneşte execuţia aici.

7 Cele trei puncte-puncte reprezintă instrucţiuni incluse în blocul care utilizează monitorul.

8 Valoarea finală a lui *n* (dacă $\neq 0$) este afişată la ieşirea din blocul utilizator.

O nouă instanţă a acestui monitor poate fi declarată local unui bloc *P*

instance *rachetă* : *contor*; *P*

În interiorul blocului *P*, procedurile care încep cu * pot fi apelate prin comenzi

rachetă.incr; ... *rachetă.decr*; ... ; **if** *rachetă.zero* **then** ...

Totuşi o procedură fără * sau variabilă, ca de exemplu *n*, nu pot fi accesate din interiorul lui *P* şi supravegherea acestei restricţii este controlată de compilator. Excluderea mutuală inerentă monitorului ne asigură că o procedură a sa poate fi apelată în siguranţă de orice număr de procese din *P* şi nu există pericolul interferării actualizării lui *n*. De observat că o încercare de a apela *rachetă.decr* când $n = 0$ va fi întârziată până ce alt proces din *P* apelează *rachetă.incr*. Aceasta ne asigură că valoarea lui *n* nu poate fi negativă niciodată.

Efectul declarării unei instanţe a monitorului este explicat printr-o variantă a regulii de copiere a unui apel de procedură în ALGOL 60. Mai întâi, se face o copie a textului monitorului, blocul utilizator. Blocul utilizator *P* este copiat în locul celor trei puncte-puncte din monitor şi toate numele locale ale monitorului sunt prefixate cu numele instanţei, cum se arată mai jos

```
rachetă.n : integer;
procedure rachetă.incr; begin rachetă.n : = rachetă.n + 1 end;
procedure rachetă.decr;
  when rachetă.n > 0 do begin rachetă.n : = rachetă.n–1 end;
function rachetă.zero : Boolean;
  begin rachetă.zero := (rachetă.n = 0) end
begin rachetă.n : = 0;
  P;
  if rachetă.n ≠ 0 then print (rachetă.n)
end
```

De observat cum regula de copiere a făcut imposibil pentru procesul utilizator să uite inițializarea valorii lui *n* sau să uite imprimarea valorii sale finale când este necesar.

Ineficiența testării repetate a condițiilor de intrare au dus la proiectarea monitoarelor după niște scheme mai elaborate pentru așteptarea explicită și semnalarea explicită a reluării proceselor în așteptare. Aceste scheme permit chiar unui apel de procedură suspendarea în mijlocul execuției sale (pe o condiție), astfel că după eliberarea automată a excluderii un apel ulterior de procedură făcut de un alt proces poate declanșa reluarea procesului suspendat. În acest fel, un număr de tehnici ingenioase de planificare pot fi eficient implementate, și totodată, acum cred că o complexitate suplimentară este pe deplin justificată.

7.2.5 Monitoare imbricate

O instanță a unui monitor poate fi privită ca un semafor pentru a proteja o singură resursă ca de exemplu o imprimantă care nu trebuie folosită de mai mult de un proces odată. Un astfel de monitor ar putea fi declarat

```
monitor resursă;
var liber : Boolean;
procedure *capturează; when liber do liber: = fals;
procedure *eliberează; begin liber : = adevărat end;
begin liber : = adevărat, ... end
```

Totuși, protecția realizată de acest monitor poate fi ocolită de un proces care utilizează resursă fără capturarea ei sau compromisă de altul care uită să o elibereze. Ambele pericole pot fi evitate printr-o construcție similară resursei virtuale (6.4 X4). Aceasta ia forma unui monitor declarat local într-un monitor – resursă prezentat mai sus. Numele resursei virtuale este notat cu * pentru a o face accesibilă declarării de către procesele utilizator. Totuși * este eliminat

din numele procedurilor ******captureazǎ* şi ******eliberează* astfel că acestea pot fi folosite numai în interiorul monitorului resursei virtuale, neputând fi folosite abuziv de alte procese

```
monitor resursă;
   liber : Boolean;
   procedure capturează;
      when liber do liber : = fals;
   procedure capturează;
      begin liber : = true end
   monitor *virtual;
      procedure *utilizează (l :linie); begin ... end;
      begin capturează; ... ; eliberează end
begin liber : = adevărat; ... end
```

O instanţă a acestui monitor este declarată astfel

```
instance imprimantă : resursă; P
```

Un bloc din *P* care cere transmisia unui fişier la o imprimantă se scrie

```
instance prog : imprimantă.virtual;
   begin ... prog.utilizează (l1); ... prog.utilizează(l2); ... end
```

Operaţiile necesare de capturare şi eliberare a imprimantei sunt automat inserate de monitorul virtual înainte şi după acest bloc utilizator într-o manieră care previne utilizarea antisocială a imprimantei. În principiu ar fi posibil pentru blocul utilizator să se împartă în mai multe procese paralele toate utilizând instanţa *prog* din monitorul virtual dar probabil că nu este aceasta intenţia. Un monitor care trebuie folosit numai pentru un singur proces este cunoscut în PASCAL PLUS ca un *plic* şi poate fi implementat mai eficient fără excludere sau sincronizare. Compilatorul verifică dacă nu este din neatenţie partajat.

Semnificaţia acestor declaraţii de instanţe poate fi înţeleasă prin aplicarea repetată a regulei de copiere cu rezultatul arătat în exemplul alăturat.

Copierea explicită arătată aici este numai pentru cei mai puţin obişnuiţi cu această tehnică. Un programator mai experimentat n-ar dori niciodată să vadă versiunea expandată sau chiar să se gândească la ea.

Aceste notaţii au fost folosite în 1975 pentru descrierea unui sistem de operare similar cu acela din paragraful 6.5. Ele au fost implementate apoi în PASCAL PLUS. Efecte extrem de ingenioase pot fi obţinute prin mixarea * cu imbricarea.

```
var imprimantă.liber : Boolean;
procedure imprimantă.capturează;
      when imprimantă.liber do imprimantă.liber : = fals;
procedure imprimantă.eliberează;begin imprimantă.liber := adevărat
                                                          end;

begin imprimantă.liber : = adevărat
      .
      .
      .
   begin
   procedure prog.imprimantă.lutilizează (l : linie); begin ... end;
      imprimantă.acaparează;
      ... prog.imprimantă.lutilizează (l1);
      ... prog.imprimantă.lutilizează (l2);
      imprimantă.eliberează;
   end;
      .
      .
      .
end
```

Cu toate acestea, notațiile din PASCAL și SIMULA par puțin stângace și explicațiile în termenii substituției și redenumirii sunt puțin cam greu de urmărit. Datorită criticilor lui Edsger W. Dijkstra asupra acestor aspecte am fost impulsionat să proiectez procesele comunicante secvențiale.

Totuși, este clar din paragraful 6.5. că partajarea ca concept implică complicații cu toate că este exprimată în cadrul teoretic al proceselor comunicante sau în cadrul regulei de copiere și semanticii apelurilor de procedură din PASCAL PLUS. Alegerea între limbaje pare a fi în parte o chestiune de gust sau eficiență. Pentru implementarea unui sistem de operare pe un calculator cu memorie principală partajată, PASCAL PLUS are probabil un avantaj.

7.2.6 Ada™

Facilitățile oferite pentru programarea concurentă în Ada sunt un amalgam de apel de procedură la distanță din PASCAL PLUS cu forma mai puțin structurată de comunicare prin intrare și ieșire. Procesele se cheamă task-uri și ele comunică printr-o declarație *call* (ca și apelurile de procedură cu parametri de ieșire și intrare) și declarații *accept* (care sunt atât în forma lor sintactică cât și în efect ca declarațiile de procedură). O declarație tipică de *accept* ar fi

accept *scrie* (*V* : **in** *integer*; *ANTER* : **out** *integer*) **do**
ANTER : = *K*; *K* := *V* **end**

Un apel corespunzător ar putea fi

scrie(37,*X*)

Identificatorul *scrie* este cunoscut ca *nume de intrare*.

O declaraţie *accept* şi una *call* cu acelaşi nume în taskuri diferite sunt executate când ambele procese sunt gata să le execute împreună. Efectul este precum urmează

(1) Parametrii de intrare sunt copiaţi prin apel la procesul care-i acceptă.
(2) Este executat corpul declaraţiei *accept*.
(3) Valorile parametrilor de ieşire sunt copiaţi înapoi prin apel.
(4) Apoi ambele taskuri îşi continuă execuţia cu următoarele lor declaraţii.

Execuţia corpului unui *accept* este cunoscută ca tehnica de *rendezvous*, deoarece atât taskul apelant cât şi cel apelat pot fi gândite că se execută împreună. *Rendezvous*-ul este o calitate atrăgătoare pentru Ada deoarece simplifică practica foarte utilizată a alternării ieşirii cu intrarea fără complicarea cazului când se foloseşte numai intrarea sau numai ieşirea.

Analogul lui Ada pentru □ este declaraţia *select* care ia forma

select
accept *citeşte* (ν : **out** *integer*) **do** ν : = *B*[*i*] **end**; *i* : = *i* + 1;...
or accept *scrie* (ν : **in** *integer*) **do** *B*[*j*] : = ν **end**; *j* : = *j* + 1;...
or ...
end select

Exact una din alternativele separate de **or** vor fi selectate pentru execuţie, depinzând de alegerea făcută de task-ul (-urile) apelant(e). Declaraţiile rămase după **end** în alternativa selectată din **accept** sunt executate la terminarea *rendezvous*-ului, concurent cu derularea taskului apelant. Selectarea unei alternative poate fi inhibată printr-o condiţie **when** falsă, de exemplu

when not *liber* ⇒ **accept** ...

Se realizează astfel efectul unei regiuni critice condiţionale.

Una din alternativele dintr-o declaraţie **select** poate începe cu un **delay** în loc de un **accept**. Această alternativă poate fi selectată dacă nici o alta nu

este selectată pe o perioadă mai mare decât un număr specificat de secunde. Scopul este de a ne proteja de pericolul ca erori hardware sau software să determine amânarea continuă a declaraţiei **select**. Deoarece modelul nostru matematic face deliberat abstracţie de timp, o întârziere nu poate fi reprezentată adecvat decât cu excepţia cazului când se permite selecţia complet nedeterministă a alternativei, începând cu întârzierea.

Una din alternative într-o declaraţie *select* poate fi **terminate**. Această alternativă este selectată când toate taskurile care ar putea apela taskul dat s-au terminat. De asemenea s-a terminat şi taskul dat. Aceasta nu este o situaţie la fel de convenabilă ca o declaraţie interioară din PASCAL PLUS care permitea monitorului să pună toate lucrurile în ordine la terminare.

O declaraţie **select** poate avea o clauză **else** care este selectată fie din cauză că nici una din celelalte alternative nu poate fi selectată imediat fie din cauză că toate condiţiile **when** sunt false fie din cauză că nu există un apel corespunzător în aşteptare în alt task. Aceasta ar putea fi ceva echivalent unei alternative cu **delay** 0.

Un apel **call** poate fi de asemenea protejat împotriva întârzierilor arbitrare printr-o declaraţie **delay** sau o clauză **else**. Aceasta ar putea conduce la ineficienţă în implementare într-o reţea distribuită de procesoare.

Taskurile în Ada sunt declarate aproape la fel ca procesele subordonate din paragraful 4.5. La fel ca monitoarele din PASCAL PLUS, fiecare task poate deservi orice număr de procese apelante. Mai mult, programatorul trebuie să se îngrijească ca taskul să se termine normal. Definiţia uni task este compusă din două părţi, specificarea şi corpul. Specificarea dă taskului numele său precum şi numele şi tipul parametrilor tuturor intrărilor prin care taskul poate fi apelat. Aceasta este informaţia utilă programatorului şi folosită de compilatorul respectiv. Corpul taskului defineşte comportarea sa şi poate fi compilat separat de programul apelant.

Fiecărui task în Ada îi poate fi atribuit o *prioritate* fixă. Dacă există mai multe taskuri decât procesoare, taskurile cu prioritate mai scăzută vor fi neglijate. Prioritatea în execuţia unui *rendezvous* este mai mare decât priorităţile taskurilor apelante sau apelate. Indicatorul de prioritate se numeşte *pragma*. Prin acest indicator se intenţionează îmbunătăţirea timpilor critici de răspuns comparaţi cu cei necritici şi nu se intenţionează afectarea comportării logice a programului. Aceasta este o idee excelentă deoarece separă preocuparea pentru corectitudinea logică abstractă de problematica răspunsului în timp real, care în general poate fi mai uşor de rezolvat printr-o judicioasă selecţie a hardware-ului sau prin experimente.

Ada oferă un număr de facilităţi adiţionale. Astfel, este posibil de testat câte apeluri aşteaptă la o intrare a unui monitor. Un task poate termina brusc alt task (*abort*) şi toate taskurile depinzând de el. Taskurile pot accesa şi actualiza variabile partajate. Aceste efecte pot fi făcute chiar mai imprevizibile prin faptul că compilatoarelor le sunt permise întârzierile, reordonările sau în-

treţeserea actualizărilor ca şi cum variabila n-ar fi fost partajată. Există de asemenea alte efecte de interacţiune şi complexităţi adiţionale cu alte caracteristici pe care nu le-am menţionat.

În afara complexităţilor arătate în paragraful precedent, multitasking-ul în Ada pare a fi mai bine proiectat pentru implementare şi utilizare pe un multiprocesor cu memorie partajată.

7.3 Comunicarea

Explorarea posibilităţilor de structurare a unui program sub formă de reţea de procese comunicante a fost printre altele motivată de dezvoltarea spectaculoasă a hardware-ului. Naşterea microprocesorului a redus rapid costul puterii de procesare cu câteva ordine de mărime. Totuşi, puterea fiecărui microprocesor individual era mult mai mică decât a unui calculator standard tradiţional şi într-un fel scump. A apărut astfel a fi mult mai economic obţinerea unei puteri mărite prin utilizarea mai multor microprocesoare care să coopereze la acelaşi task. Aceste microprocesoare puteau fi conectate ieftin prin canale de-a lungul cărora să comunice între ele. Fiecare microprocesor putea avea memoria sa principală locală pe care să o acceseze la viteza maximă, evitând astfel blocajele ce ar rezulta când mai multe microprocesore accesează la un moment dat o memorie partajată.

7.3.1 Conducte

Cel mai simplu model de comunicaţie între elemente de procesare este transferul unidirecţional al mesajului între fiecare proces şi vecinul său, ceea ce constituie o conductă (*pipe*) liniară ca în paragraful 4.4. Ideea a fost prima dată pusă în discuţie de Conway care a ilustrat-o prin exemple similare celor din 4.4 X2 şi X3 cu deosebirea că se aştepta ca cele două procese componente ale conductei să se termine cu succes în loc să cicleze infinit. El a propus modalitatea de a fi folosită structura de conductă pentru scrierea unui compilator în mai mulţi paşi pentru un limbaj de programare. Astfel, pe un calculator cu o memorie principală adecvată, toţi paşii sunt activi simultan în memorie şi controlul este trecut pe rând între paşi împreună cu transferul de mesaje, simulând astfel execuţia paralelă. Pe un calculator cu mai puţină memorie principală, numai un pas este activ o dată el trimiţându-şi ieşirea într-un fişier din memoria secundară. La terminarea fiecărui pas, începe următorul pas care îşi preia intrarea din fişierul produs de predecesorul său. Totuşi, rezultatul final al compilatorului este exact acelaşi în ciuda diferenţelor radicale în problema execuţiei. Este caracteristic pentru abstractizarea cu succes în programare că implementarea poate fi făcută prin mai multe metode care la rândul lor sunt eficiente funcţie de diferite circumstanţe. În acest caz sugestia lui

Conway ar putea fi foarte prețioasă pentru implementări software pe o gamă de calculatoare, oferind o gamă largă de opțiuni pentru mărimea memoriei.

Conceptul de conductă este de asemenea metoda standard de comunicație în sistemul de operare UNIX™, unde notația "|" corespunde lui "≫" de la noi.

7.3.2 Canale bufferate multiple

Modelul conductei permite unui lanț liniar de procese să comunice numai într-o singură direcție și nu contează dacă secvența de mesaje este bufferată sau nu. Generalizarea naturală a conductei este de a permite fiecărui proces să comunice cu orice alt proces în orice direcție. La prima vedere, pare aproape natural să asigurăm în aceste condiții bufferarea pe toate canalele de comunicație. În proiectarea sistemului de operare RC4000, facilitatea de comunicare bufferată a fost implementată în kernel și a fost utilizată pentru comunicarea între module asigurând servicii către un nivel superior. La o scară mai mare, o rețea de memorare și comutare a pachetelor, ca de exemplu ARPAnet în USA, interpune inevitabil buffere între sursa și destinația mesajelor.

Când modelul de comunicație între procese este generalizat de la un lanț liniar la o rețea care poate fi și ciclică, prezența sau absența bufferării poate determina o diferență vitală privind comportarea logică a sistemului. Prezența bufferării nu este totdeauna favorabilă : de exemplu, este posibil de a scrie un program care se poate bloca dacă lungimea bufferului este permis să depășească cinci, tot așa cum un program diferit se va bloca dacă lungimea bufferului nu este permis să depășească cinci. Pentru a evita astfel de iregularități, lungimea tuturor bufferelor ar putea fi infinită. Din nefericire, aceasta ne conduce la grave probleme de eficiența implementării când memoria principală este umplută cu buffere de mesaje. Tratarea matematică este de asemenea complicată prin faptul că orice rețea este un automat cu un număr infinit de stări, chiar când procesele componente sunt în număr finit. În fine, pentru o rapidă și controlabilă interacțiune între oameni și calculatoare, bufferele trebuie afectate numai canalelor, deoarece presupun o întârziere între stimul și răspuns. Dacă ceva nu e în regulă în procesarea unui stimul bufferat, este mult mai dificil de urmărit eroarea și de eliminat. Bufferarea este o tehnică de procesare de tip batch și ar trebui evitată ori de câte ori interacțiuni rapide sunt mai importante decât utilizarea intensivă a unui procesor.

7.3.3 Multiprocesare funcțională

Un proces determinist poate fi definit în termenii unei funcții matematice de la canalele sale de intrare la cele de ieșire. Fiecare canal este identificat cu secvența infinit extensibilă de mesaje care trece de-a lungul lui. Astfel de funcții sunt definite în general recursiv folosind structura secvențelor de in-

trare, mai puțin cazul unei secvențe de intrare vide. De exemplu, procesul care emite rezultatul multiplicării fiecărui număr de intrare cu un n este definit

$$prod_n(stânga){=}{<}n\times stânga_0{>}\hat{}\,prod_n\,(stânga')$$

O funcție care are două șiruri sortate (strict) ca parametri și emite șirul comun sortat (strict) este definită

$dup2(stg1,stg2)=$
  if $stg1_0 < stg1_0$ **then** $<stg1_0>\hat{}\,dup2(stg1',stg2)$
    else if $stg2_0 < stg1_0$ **then** $<stg2_0>\hat{}\,dup2(stg1,stg2')$
      else $<stg2_0>\hat{}\,dup2(stg1',stg2')$

O rețea neciclică poate fi reprezentată printr-o compunere de astfel de funcții. De exemplu, o funcție care reunește trei șiruri sortate de intrare poate fi definită ca

$$dup3(stg1,stg2,stg3){=}dup2(stg1,dup2\,(stg2,stg3))$$

Figura 7.1 prezintă diagrama de conexiune a acestei funcții.

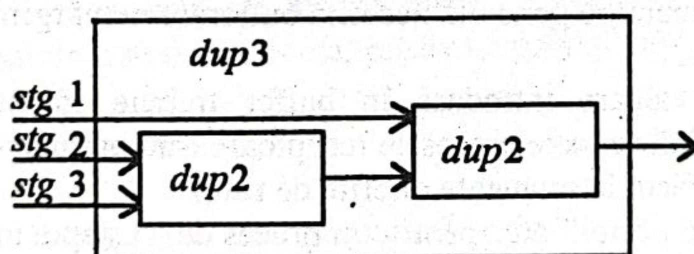


Figura 7.1

O rețea ciclică poate fi construită cu ajutorul unui set de ecuații recursive mutual. De exemplu, considerăm problema atribuită de Dijkstra lui *Hamming* și anume de a defini o funcție care generează într-o secvență crescătoare toate numerele care au numai pe 2, 3 și 5 ca factori netriviali. Primul astfel de număr este 1. Dacă x este un astfel de număr sunt și $2\times x$, $3\times x$ și $5\times x$ de asemenea. Pentru problema noastră utilizăm trei procese $prod_2$, $prod_3$ și $prod_5$ pentru a genera aceste produse, rezultatele alimentând procesul *dup*3, care ne asigură că sunt sortate într-o ordine crescătoare (fig. 7.2).

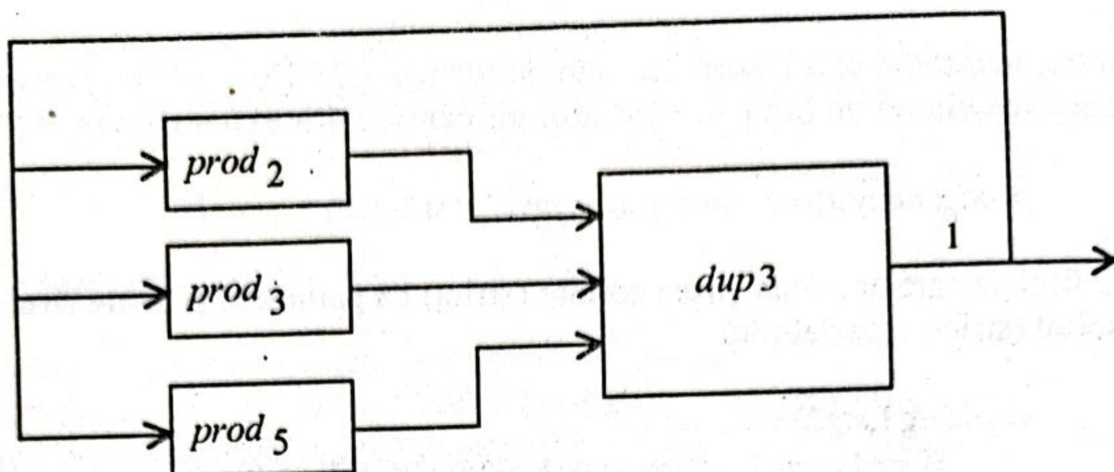


Figura 7.2

Funcţia care generează rezultatul dorit nu are intrări. Ea este definită simplu de

Hamming=
$<1>$^$dup3(prod_2(Hamming), prod_3(Hamming), prod_5(Hamming))$

Metoda funcţională pentru o reţea de procesoare este foarte diferită de cea dată în această carte în următoarele aspecte

(1) O implementare generală necesită bufferare nemărginită pe toate canalele.
(2) Fiecare valoare introdusă în buffer trebuie să fie reţinută în buffer până ce toate procesele receptoare l-au preluat, ceea ce poate fi făcut la momente diferite de timp.
(3) Nu există posibilitatea pentru un proces de a aştepta una din două intrări, oricare ar sosi prima.
(4) Procesele sunt toate deterministe.

Cercetări recente au dus la reducerea ineficienţei condiţiilor (1) şi (2) şi la relaxarea restricţiilor (3) şi (4).

7.3.4 Comunicaţie nebufferată

De mulţi ani, eu am decis să aleg comunicaţia nebufferată (sincronizată) ca un lucru de bază. Motivele au fost

(1) Corespunde îndeaproape conexiunilor directe ale procesoarelor (prin fire). Firele nu pot reţine mesaje.
(2) Corespunde îndeaproape efectului de apel şi întoarcere din subrutină pe un singur procesor, prin copierea valorile parametrilor şi rezultatelor.

(3) Când se doreşte bufferarea, poate fi implementată simplu ca un proces, iar gradul de bufferare poate fi precis controlat de programator.
(4) Alte dezavantaje ale bufferelor au fost menţionate la sfârşitul paragrafului 7.3.2.

Desigur, nici unul din aceste argumente nu este pe deplin convingător. De exemplu, dacă comunicarea bufferată ar fi fost luată ca primitivă nu s-ar fi putut face nici o diferenţă logică în cazul banal al alegerii între un apel şi un return dintr-o subrutină. Sincronizarea poate fi realizată în toate celelalte cazuri prin succedarea fiecărei ieşiri cu recepţia unei confirmări, iar a oricărei intrări prin emiterea unei confirmări.

7.3.5 Procese secvenţiale comunicante

Acesta a fost titlul primei mele expuneri despre un limbaj de programare bazat pe concurenţă şi comunicare. Respectiva propunere diferă de această carte în două aspecte semnificative.

(1) Compunere paralelă

Canalele nu sunt numite. În loc de numire, procesele componente ale unei construcţii paralele au nume unice, prefixate prin ::

$*[a::P \| b::Q \| \ldots \| c::R]$

În procesul P comanda $b!v$ generează valoarea v procesului etichetat b. Această valoare este recepţionată printr-o comandă $a?x$ intervenind în procesul Q. Numele proceselor sunt locale comenzii paralele în care sunt introduse şi comunicaţiile între procesele concurente sunt mascate.

Avantajul acestei scheme rezidă în faptul că nu mai este nevoie de a introduce în limbaj vreun concept de canal sau o declaraţie de canal. Mai mult, este logic imposibil de violat restricţia ca un canal să existe între mai mult de două procese iar unul dintre ele să-l utilizeze pentru intrare şi altul pentru ieşire. Dar sunt şi câteva dezavantaje atât în practică cât şi în teorie.

(1) Un dezavantaj practic serios este acela că un proces subordonat trebuie să ştie numele procesului care-l utilizează, aceasta complicând construcţia bibliotecii de procese subordonate.
(2) Un dezavantaj din punct de vedere matematic este că operatorul de compunere paralelă are un număr variabil de parametri şi nu poate fi redus la un operator binar asociativ, ca $\|$.

(2) Terminare automată

În prima versiune, toate procesele dintr-o compunere paralelă trebuiau să se termine. Motivul consta în speranţa ca corectitudinea unui proces să poată fi specificată în acelaşi fel ca *post-condiţia* pentru un program convenţional, cu alte cuvinte printr-un predicat adevărat la terminarea cu succes. (Speranţa nu s-a împlinit şi alte metode de demonstrare (paragraful 1.10) par acum mai satisfăcătoare). Obligaţia ca un proces subordonat să se termine determină obligaţia deranjantă pentru procesul care-l utilizează de a semnala terminarea sa tuturor subordonaţilor. De aceea s-a introdus o convenţie *ad-hoc*. O buclă de forma

$$*[a?x \rightarrow P \Box b?x \rightarrow Q \Box \ldots]$$

se termină automat la terminarea tuturor proceselor conectate la *a*, *b*, ... de la care se cere un mesaj de intrare. Aceasta permite procesului subordonat de a termina orice secvenţă de cod necesară înaintea terminării propriu-zise, o caracteristică care s-a dovedit utilă în SIMULA şi PASCAL PLUS.

Problema cu această convenţie constă în complexitatea definirii şi implementării. Metodele de a demonstra corectitudinea unui program nu par mai simple cu ea decât fără. Acum îmi apare mai normal (ca în paragraful 4.5) de a relaxa restricţia ca un proces subordonat să trebuiască să se termine, în cazuri mai complicate trebuind să fie luate alt gen de măsuri (paragraful 6.4).

7.3.6 Occam

În contrast cu Ada, Occam este un limbaj de programare foarte simplu şi care urmăreşte îndeaproape principiile expuse în această carte. Cele mai evidente şi contrastante deosebiri sunt notaţionale. Sintaxa Occam este proiectată pentru a fi compusă direct pe ecran cu ajutorul unui editor conector de sintaxă, folosind operatori prefix în loc de sufix şi indentarea în locul parantezelor.

```
SEQ
    P                                   pentru (P;Q;R)
    Q
    R

PAR
    P                                   pentru (P||Q||R)
    Q
    R
```

```
ALT                                         pentru (c?x→P□d?y→Q)
    c?x
      P
    d?y
      Q

IF                                          pentru (P◁B▷Q)
    B
      P
    NOT B
      Q
    WHILE B                                 pentru (B*P)
    P
```

Construcţia *ALT* corespunde declaraţiei *select* din Ada şi oferă o gamă similară de opţiuni. Selecţia unei alternative poate fi inhibată prin falsitatea unei condiţii boolene *B*

```
B & c?x
  P
```

Această intrare poate fi înlocuită printr-un *SKIP*, caz în care alternativa poate fi selectată ori de câte ori garda booleană este adevărată sau poate fi înlocuită printr-o aşteptare care permite alternativei să fie selectată după scurgerea unui interval specific. Limbajul Occam nu are notaţii distincte pentru conducte (paragraful 4.4), procese subordonate (paragraful 4.5) sau procese partajate (paragraful 6.4). Toate situaţiile de comunicaţie trebuie realizate prin identitatea explicită a numelor canalelor. Pentru a veni în ajutor, procedurile pot fi declarate cu parametri canale. Procesul simplu de copiere poate fi declarat

```
PROC copie(CHAN stg,dr) =
    WHILE TRUE
    VAR x:
    SEQ
        stg?x
        dr!x:
```

Bufferul dublu *COPIE≫COPIE* poate fi acum construit

```
CHAN mij:
PAR
    copie(stg,mij)
    copie(mij,dr)
```

Un lanţ de *n* buffere poate fi construit utilizând un masiv de *n* canale şi o formă iterativă de construcţie paralelă care construieşte *n*-2 procese, una pentru fiecare valoare a lui *i* între 0 şi *n*-3 inclusiv

```
CHAN mij[n-1]:
PAR
        copie(stg,mij[0])
        PAR i=[0 FOR n-2]
         copie (mij[i], mij[i+1] )
         copie(mij[n-2],dr)
```

Din cauză că Occam se intenţionează a fi implementat cu alocare statică de memorie la un număr fixat de procesoare, valoarea lui *n* în exemplul de mai sus trebuie să fie o constantă. Din acelaşi motiv, procedurile recursive nu sunt permise.

O construcţie similară poate fi utilizată pentru a realiza efectul unor procese subordonate, de exemplu

```
PROC dublu(stg,dr)=
    WHILE TRUE
     VAR x:
     SEQ
          stg?x
          dr!(x+x):
```

Acestă construcţie poate fi declarată subordonată unui singur proces *P* care o utilizează

```
CHAN dub.stg,dub.dr:
PAR
    dublu(dub.stg,dub.dr)
    P
```

În *P* un număr este dublat prin

```
dub.stg!4;dub.dr?y; ...
```

Procesele pot fi partajate utilizând masive de canale (cu câte un element pentru fiecare proces utilizator) şi o formă iterativă a construcţiei *ALT*. De exemplu, să luăm un integrator care după fiecare nouă recepţie transmite suma tuturor elementelor pe care le-a recepţionat până atunci

```
CHAN adună[n], integral[n]:
PAR
    VAR suma,x:
    SEQ
    suma:=0
    ALT i:=[0 FOR n]
        adună[i]?x
        SEQ
        suma:=suma+x
        integral[i]!suma
    PAR i=[0 FOR n]
    ...procese utilizator...
```

Ca şi în Ada, Occam permite programatorului asignarea de priorităţi relative unor procese compuse în paralel. Acesta se face utilizând *PRI PAR* în loc de *PAR*. Procesele din capul listei au priorităţi mai mari. Facilităţile de editare pe ecran asigurate de limbaj permit reorganizarea proceselor când e necesar. O opţiune similară este oferită pentru construcţia *ALT* de *PRI ALT*. Aceasta permite pentru mai mult de o alternativă gata pentru selecţie imediată, alegerea primei textual apărute – altfel efectul este acelaşi ca şi pentru *ALT*. Desigur, programatorul este rugat să se asigure că programele sale sunt corecte logic, independent de asigurarea priorităţilor.

Există de asemenea facilităţi pentru distribuirea proceselor între procesoare distincte şi pentru specificarea pinilor fizici de la fiecare procesor ce vor fi folosiţi pentru fiecare canal relevant din programul Occam precum şi pinilor folosiţi pentru încărcarea codului programului însuşi.

7.4 Modele matematice

Recunoaşterea ideii că un limbaj de programare ar trebui să aibă o semnificaţie matematică precisă sau semantică datează de la începuturile lui 1960. Aspectul matematic asigură o specificare sigură, fără ambiguităţi, precisă şi stabilă a limbajului pentru a servi ca interfaţă agreabilă între utilizatori şi implementatori. Mai mult, de aici derivă motivele sigure pentru dezideratul ca implementări diferite să fie implementări ale aceluiaşi limbaj. Astfel, semanticile matematice sunt la fel de importante pentru obiectivul de standardizare a limbajului ca şi măsurătorile şi calculele la standardizarea şuruburilor şi piuliţelor.

După 1960 a fost recunoscut rolul important al semanticilor matematice şi anume acela de a ajuta programatorul să se debaraseze de obligaţia stabilirii corectitudinii programului său. Într-adevăr, R.W. Floyd a sugerat ca semanticile să fie formulate mai bine ca o mulţime de reguli de demonstrare

(axiome) decât ca un model matematic explicit. Această sugestie a fost adoptată în specificarea limbajelor PASCAL, Euclid și Gypsy.

Prima proiectare a Proceselor Secvențiale Comunicante (paragraful 7.3.5) nu avea o semantică matematică și lăsa deschise un număr de întrebări de proiectare ca de exemplu :

(1) Este posibil de a imbrica o comandă paralelă în alta ?
(2) Dacă da, este posibil de a scrie o procedură recursivă care se apelează în paralel ?
(3) Este teoretic posibil de a folosi comenzi de ieșire în gărzi ?

Modelul matematic dat în această carte răspunde cu "Da" la toate aceste întrebări.

7.4.1 Calculul sistemelor comunicante

O deschidere majoră în modelarea matematică a concurenței a fost făcută de Robin Milner. Obiectivul major al investigației sale a fost asigurarea unui cadru pentru construcția și compararea diferitelor modele la diferite nivele de abstractizare. Astfel, el a început cu sintaxa de bază a expresiilor dedicate să semnifice procese și a definit o serie de echivalențe între expresii din care cele mai importante sunt :

- echivalență tare
- echivalență observațională
- congruență observațională

Fiecare echivalență definește un model diferit de concurență. Inițialele CCS se referă de obicei la modelul obținut prin adoptarea congruenței observaționale ca definiție a egalității între procese.

Notațiile de bază din CCS sunt ilustrate prin corespondența următoare

$a.P$	corespunde la $a \rightarrow P$
$(a.P) + (b.Q)$	corespunde la $(a \rightarrow P \mid b \rightarrow Q)$
NIL	corespunde la $STOP$

Mai importante decât aceste diferențe notaționale sunt distincțiile în tratarea mascării. În CCS există un simbol special τ care semnifică apariția unui eveniment mascat sau o tranziție internă. Avantajul păstrării acestei înregistrări trecute a unui eveniment mascat este acela că poate fi utilizat fără restricții în gărzile ecuațiilor recursive asigurându-ne astfel că au soluții unice, așa cum s-a arătat în paragraful 2.8.3. Un al doilea (dar poate mai puțin semnificativ) avantaj este acela că procesele care se angajează într-o secvență

nemărginită de τ-uri nu se reduc toate la *CHAOS*. Astfel, distincţii utile pot fi făcute între procese divergente. Totuşi CCS eşuează în distingerea unui proces posibil divergent de unul care este similar în comportare dar nu-i divergent. Este de aşteptat ca aceasta să determine imposibilitatea unei implementări eficiente pentru limbajul CCS.

CCS nu include $\sqcap$ ca operator primitiv. Totuşi, nedeterminismul poate fi implementat prin utilizarea lui τ, de exemplu

$(\tau.P)+(\tau.Q)$	corespunde la $(P \sqcap Q)$
$(\tau.P)+(a.Q)$	corespunde la $P \sqcap (P \square (a \rightarrow Q))$

Dar aceste corespondenţe nu sunt exacte din cauză că în CCS nedeterminismul definit de τ nu este asociativ aşa cum rezultă din faptul că arborii din fig. 7.3. sunt distincţi.

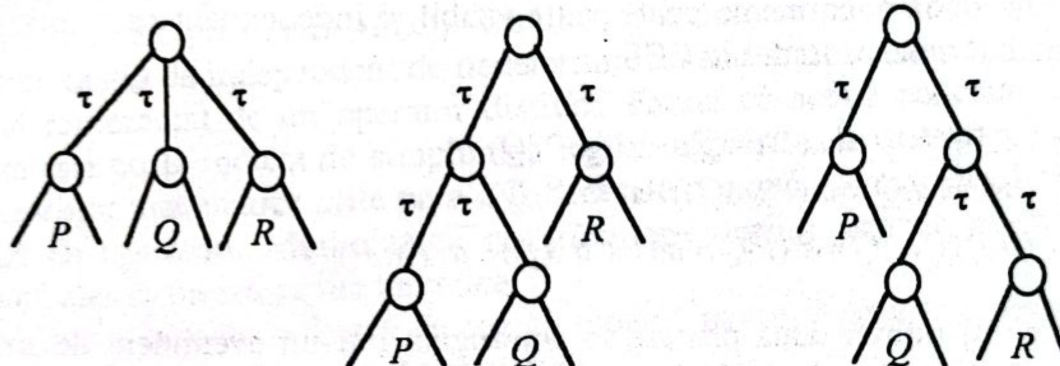


Figura 7.3

Mai mult, prefixarea nu se distribuie cu nedeterminismul, din cauză că arborii din fig. 7.4 sunt distincţi când $P \neq Q$

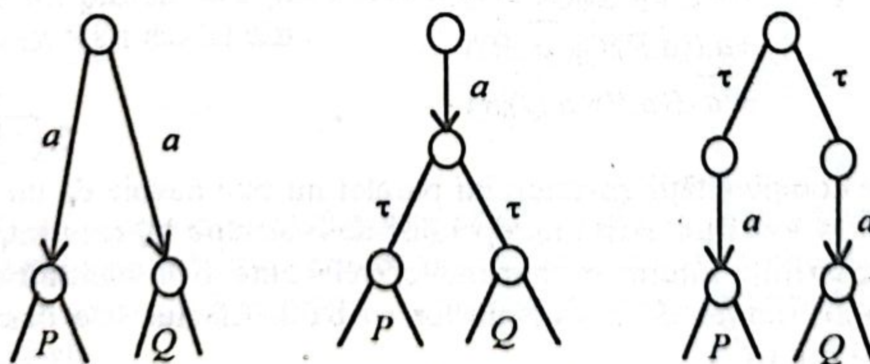


Figura 7.4

Aceste exemple ne arată că CCS face multe distincţii între procese care ar putea fi privite ca identice în această carte. Motivul pentru aceasta este că CCS se intenţionează a fi un cadru pentru o familie de modele, fiecare din ele putând face mai multe identificări decât CCS, dar nu mai puţine. Pentru a

evita restrângerea gamei de modele, CCS face numai acele identificări care par a fi absolut necesare. În modelul matematic din această carte am urmărit exact scopul opus – am făcut cât mai multe identificări posibile, păstrând numai distincțiile cele mai importante. De aceea avem o mulțime mult mai mare de legi algebrice. Se speră că aceste legi vor fi de utilitate practică în raționamente de proiectare și în implementări. În particular ele permit mai multe transformări și optimizări decât CCS.

Operatorul de concurență de bază din CCS se notează prin |. El este cu mult mai complicat decât operatorul ||, în el incluzându-se aspecte de mascare, nedeterminism, întrețesere ca și sincronizare. Fiecare eveniment din CCS are două forme, simplă (a) sau barată ($\bar{a}$). Când două procese sunt reunite pentru a evolua concurent, sincronizarea apare numai când un proces se angajează în evenimentul cu bară și celălalt proces se angajează în evenimentul corespunzător simplu. Participarea lor comună la un astfel de eveniment este mascată prin conversia imediată la τ. Totuși, sincronizarea nu este obligatorie. Fiecare din cele două evenimente poate apare vizibil și independent ca o interacțiune cu mediul extern. Astfel în CCS

$$(a.P)|(b.Q)=a.(P|(b.Q))+b.((a.P)|Q)$$
$$(a.P)|(a.Q)=a.(P|(a.Q))+a.((a.P)|Q)$$
$$(a.P)|(\bar{a}.Q)=\tau.(P|Q)+a.(P|(\bar{a}.Q))+\bar{a}.((a.P)|Q)$$

Mai mult, numai două procese se pot angaja într-un eveniment de sincronizare. Dacă mai mult decât două procese sunt gata, alegerea unei perechi este nedeterministă

$$\begin{aligned}(a.P)|(a.Q)|(\bar{a}.R)=&\tau.(P|(a.Q)|R)+\tau.((a.P)|Q|R)\\ &+a.(P|(a.Q)|(\bar{a}.R))\\ &+a.((a.P)|Q|(\bar{a}.R))\\ &+\bar{a}.((a.P)|(a.Q)|R)\end{aligned}$$

Din cauza complexității operatorului paralel nu este nevoie de un operator de mascare. În locul lui există un operator de *restricție* \ , care împiedică pur și simplu aparițiile tuturor evenimentelor relevante și le elimină din alfabetul procesului, împreună cu varianta lor cu bară. Efectul este ilustrat de următoarele legi din CCS

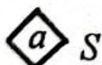
$$(a.P)\backslash\{a\} \quad =(\bar{a}.P)\backslash\{a\} \quad =NIL$$
$$(P+Q)\backslash\{a\} \quad =(P\backslash\{a\})+(Q\backslash\{a\})$$
$$((a.P)|(\bar{a}.Q))\backslash\{a\} \quad =\tau.((P|Q)\backslash\{a\})$$

$$((a.P)|(a.Q)|(\overline{a}.R))\backslash\{a\}=\tau.((P|(a.Q)|R)\backslash\{a\})$$
$$+\tau.(((a.P)|Q|R)\backslash\{a\})$$

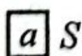
Ultima lege de mai sus ilustrează puterea operatorului paralel din CCS în realizarea efectului de partajare a procesului ($\overline{a}.R$) între două procese utilizator ($a.P$) şi ($a.Q$). Unul din obiectivele CCS-ului a fost de a realiza maximum de putere de expresie cu cât mai puţini operatori distincţi posibili. Aceasta este sursa eleganţei şi puterii CCS-ului şi într-adevăr simplifică investigaţia familiilor de modele definite de diferite relaţii de echivalenţă.

În această carte s-a ales o cale complementară. Simplitatea este văzută prin proiectarea unui model simplu, în termenii căruia este uşor de definit cât mai mulţi operatori potriviţi pentru investigarea unei game de concepte distincte. De exemplu, alternativa nedeterministă $\sqcap$ introduce nedeterminismul în forma sa cea mai pură şi este cât se poate de independentă de controlul mediului reprezentat prin ($x{:}B \rightarrow P(x)$). Similar, $\|$ introduce concurenţa şi sincronizarea destul de independent de nedeterminism sau mascare, fiecare dintre ele fiind reprezentat de un operator distinct. Faptul că aceste concepte sunt distincte este poate indicat de simplitatea legilor algebrice. În aplicarea practică a teoriilor matematice utile pare a fi necesară o gamă cât mai largă şi rezonabilă de operatori. Minimizarea mulţimii operatorilor este de asemenea utilă, mai ales în investigaţiile teoretice.

Milner a introdus o formă de logică modală pentru a specifica comportarea observabilă a unui proces. Modalitatea

 S

descrie un proces care poate efectua a şi apoi să se comporte aşa cum este descris de S iar dualul său

 S

descrie un proces care dacă porneşte cu a trebuie să se comporte la fel ca S. Este definit un calcul de corectitudine care permite dovedirea faptului că un proces P satisface specificaţia S, un fapt care este exprimat în notaţia logică tradiţională

$$P \models S$$

Calculul este foarte diferit de acela din relaţia sat, din cauză că se bazează mai mult pe structura specificării decât pe structura programelor. De exemplu, regula pentru negaţie este

Dacă *nu* este adevărat că $P \models F$
atunci $P \models \neg F$

Aceasta înseamnă că tot procesul P trebuie scris înaintea pornirii demonstrării corectitudinii sale. În contrast, utilizarea lui **sat** permite ca demonstrarea corectitudinii unui proces component să fie construită din demonstrarea corectitudinii părților sale. Logica modală este un subiect de mare interes teoretic, dar în contextul proceselor comunicante nu a arătat încă mare lucru pentru o aplicare utilă.

În general, egalitatea în CCS este o relație tare deoarece procese egale trebuie să semene între ele atât în comportarea lor observabilă cât și în structura comportării lor mascate. De aceea CCS este un model bun pentru formularea și explorarea variatelor definiții slabe de echivalență care ignoră diversele aspecte ale comportării ascunse. Milner a realizat aceasta prin introducerea conceptului de echivalență observațională. Aceasta implică definirea unei mulțimi de observații sau evenimente (experimente) care pot fi făcute cu un proces. Două procese sunt echivalente dacă nu există nici o observație care poate fi făcută pentru unul și nu poate fi făcută pentru celălalt – o frumoasă aplicare a principiului filozofic de identitate a indescifrabilului. Principiul a fost luat ca bază a teoriei matematice din carte, care echivalează un proces cu mulțimea observațiilor care pot fi făcute asupra comportării sale. Un semn al succesului principiului este acela că două procese P și Q sunt echivalente dacă și numai dacă ele satisfac aceeași specificație

$$\forall S, P \models S \equiv Q \models S$$

Din nefericire, nu totdeauna este la fel de simplu. Dacă două procese trebuie privite ca egale, rezultatul transformării lor prin aceeași funcție ar trebui să fie de asemenea egal.

$$(P \equiv Q) \Rightarrow (F(P) \equiv F(Q))$$

Dacă τ se presupune că este mascat, o definiție naturală a unei observații ar putea conduce la echivalența

$$(\tau.P) \equiv P$$

Totuși $(\tau.P + \tau.NIL)$ n-ar trebui să fie echivalent cu $(P + NIL)$, care este egal cu P, deoarece primul poate face o alegere nedeterministă către blocaj în loc să se comporte ca P.

Soluția lui Milner la această problemă este utilizarea *congruenței* în locul echivalenței. Între experimentele care pot fi realizate cu procesul P unul este acela de a-l așeza într-un mediu $F(P)$ (unde [illegible] este compusă din alte

procese cu ajutorul operatorilor limbajului) şi apoi de a observa comportarea ansamblului. Procesele P şi Q sunt (din punct de vedere observabil) congruente dacă pentru fiecare F exprimată în limbaj, procesul $F(P)$ este echivalent observabil cu $F(Q)$. Potrivit acestei definiţii, $\tau.P$ nu este congruent cu P. Descoperirea unei mulţimi întregi de legi de congruenţă este o realizare matematică semnificativă.

Necesitatea unei complexităţi suplimentare în cazul congruenţei observaţionale se datorează imposibilităţii de a face observaţii suficient de penetrante asupra comportării lui P fără plasarea lui într-un mediu $F(P)$. De aceea, noi am fost nevoiţi să introducem conceptul de *mulţime* de refuz faţă de refuzul unui singur eveniment. Mulţimea de refuz pare a fi cel mai slab fel de observaţie care reprezintă eficient posibilitatea blocajului nedeterminist, de aceea ne conduce la o echivalenţă mai slabă şi la o mulţime mult mai puternică de legi decât în CCS.

Descrierea dată mai sus a subliniat pe larg diferenţele faţă de CCS şi a stabilit încă o dată spectrul aplicaţiilor practice ale metodei prezentată în această carte. Cele două metode au în comun o caracteristică foarte importantă şi anume o bază matematică promiţătoare pentru raţionamente despre specificaţii, modelare şi implementare. Fiecare din ele poate fi utilizată atât pentru investigaţii teoretice cât şi pentru aplicaţii practice.

Bibliografie selectivă

Conway, M.E. "Design of a Separable Transition-Diagram Compiler", *Comm. ACM* **6** (7), 8-15 (1963)
Articol clasic despre corutine

Hoare, C.A.R. "Monitors: An Operating System Structuring Concept", *Comm. ACM* **17** (10), 549-557 (1974)
Un concept de programare menit să ajute la construcţia sistemelor de operare

Hoare, C.A.R. "Communicating Sequential Processes", *Comm. ACM* **21** (8), 666-677 (1978)
Un model de programare - o versiune anterioară celei prezentate în această carte

Milner, R. *A Calculus of Communicating Systems*, Lectures Notes in Computer Science 92, Springer Verlag, New-York (1980)
O tratare matematică clară a teoriei generale despre concurenţă şi sincronizare

Kahn, G. "The Semantics of a simple language for Parallel Programming", în *Information Processing*, 74, North Holland, Amsterdam pp.471-475 (1974)
O tratare elegantă a multiprogramării funcţionale

Welsh, J. şi McKeag, R.M. *Structured System Programming*, Prentice-Hall London, pp. 324 (1980)
O trecere în revistă a lui PASCAL PLUS şi a utilizării sale în structurarea unui sistem de operare şi a unui compilator

Filman, R.E. şi Friedman, D.P. *Coordinated Computing, Tools and Techniques for Distributed Software*, McGraw-Hill pp. 370 (1974)
O utilă trecere în revistă cu trimiteri bibliografice

Dahl, O-J. "Hierarchical Program Structures", în *Structured Programming*, Academic Press, London pp. 175-220 (1982)
O introducere la ideile de bază ale lui SIMULA 67

(INMOS Ltd.) *occam™ Programming Manual* Prentice-Hall International, pp. 100 (1984)

(ANSI/MIL-STD 1815A) *Ada™ Reference Manual*
Capitolul 9 descrie caracteristica de tasking

Brookes, S.D., Hoare, C.A.R. şi Roscoe, A.W. "A Theory of Communicating Sequential Processes", *Journal ACM* **31** (7), 560-599 (1984)
O trecere în revistă a aspectelor matematice ale proceselor nedeterministe, dar fără divergenţe

Brookes, S.D. şi Roscoe, A.W. "An Improved Failures Model for Communicating Sequential Processes", în *Proceedings NSF-SERC Seminar on Concurrency*, Springer Verlag, New-York, Lectures Notes in Computer Science (1985)
Îmbunătăţirea este în legătură cu divergenţele

INDEX

S

T

U

V